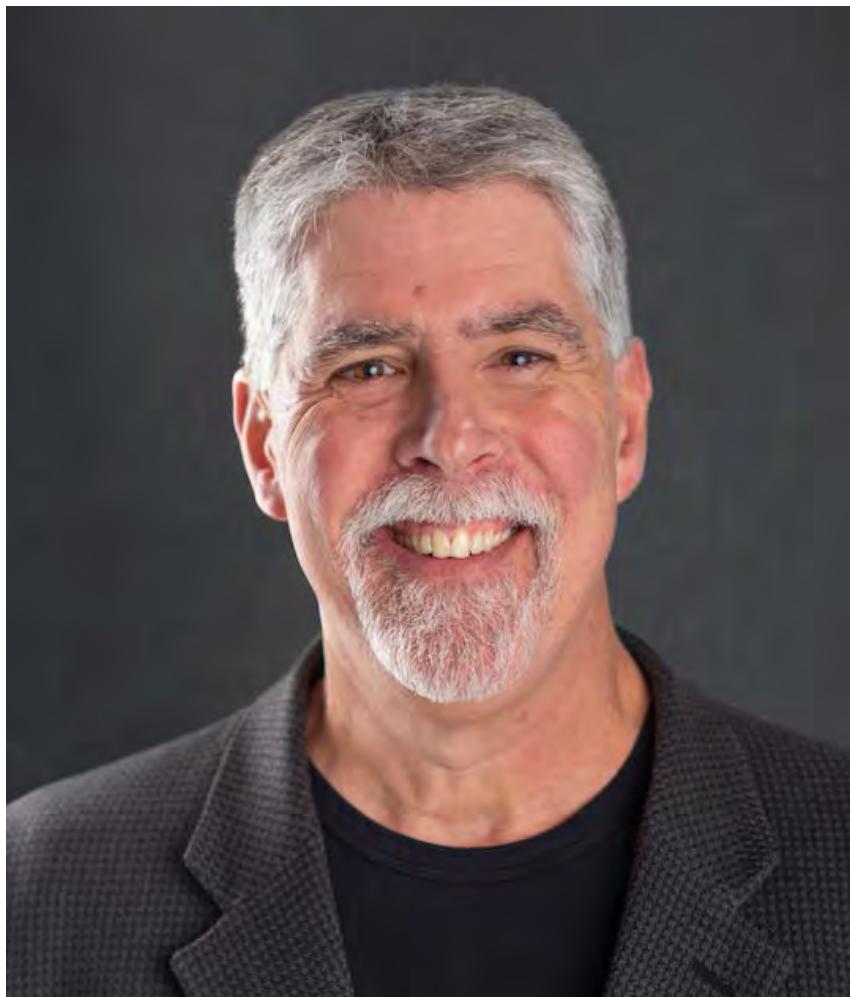




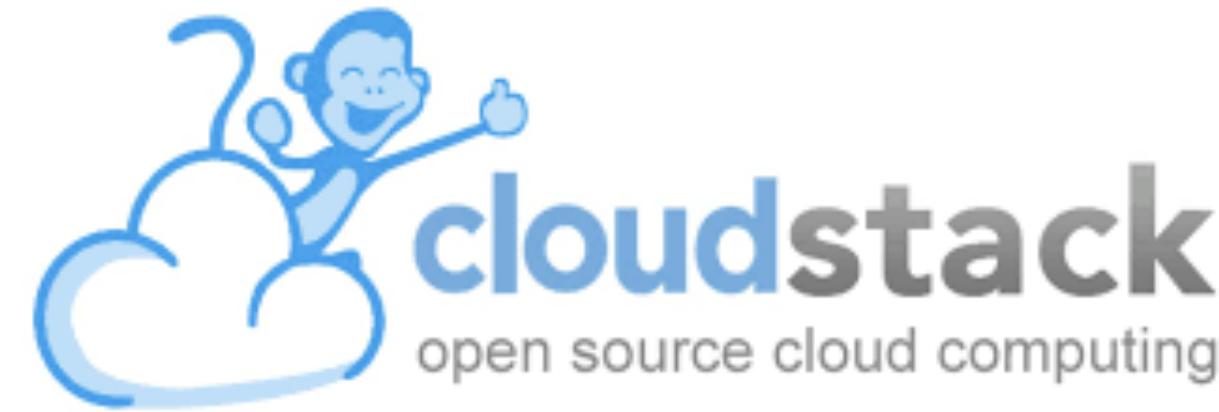
# Microservices Architecture and Design



**Mark Richards**  
**Independent Consultant**  
Hands-on Software Architect / Published Author  
Founder, [DeveloperToArchitect.com](#)  
<https://www.linkedin.com/in/markrichards3>  
[@markrichardssa](https://twitter.com/markrichardssa)



Apache ZooKeeper™



kubernetes  
by Google™





## Goals



The **desired results** plans and commits to achieve the end toward which effort is establishing specific, realistic time-targeted objectives.

gain a full understanding of microservices from an architecture and design standpoint

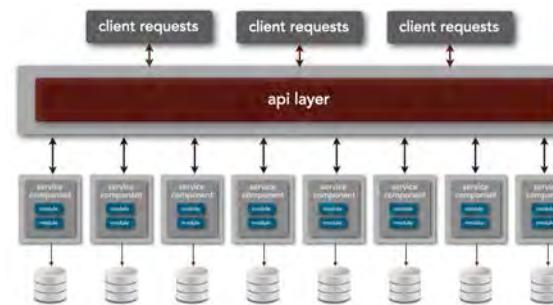
gain an understanding of the implications and challenges of the microservices architecture style

validate whether microservices is the right choice for your company and understand the alternatives

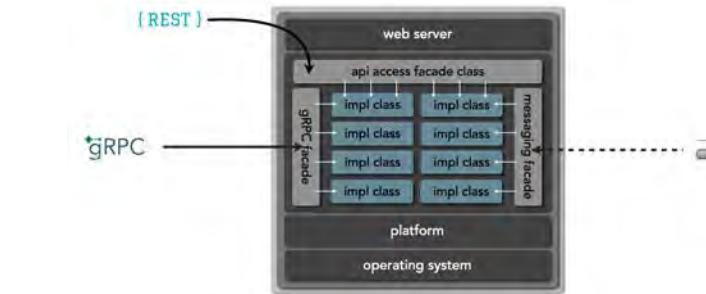
# course agenda



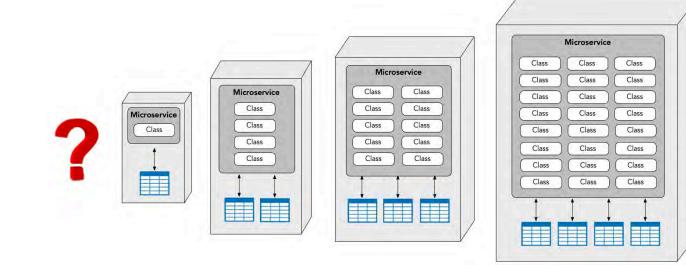
architectural  
modularity



microservices  
core concepts



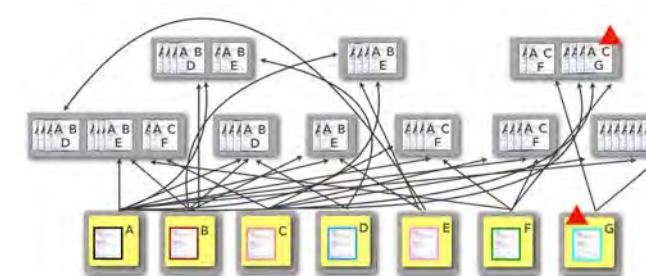
service  
design



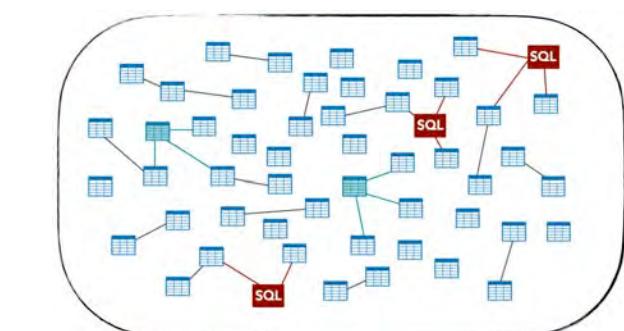
service  
granularity

[**FunctionalService**]  
[**OrchestrationService**]  
[**AggregationService**]  
[**AdapterService**]  
[**InfrastructureService**]  
[**DataService**]

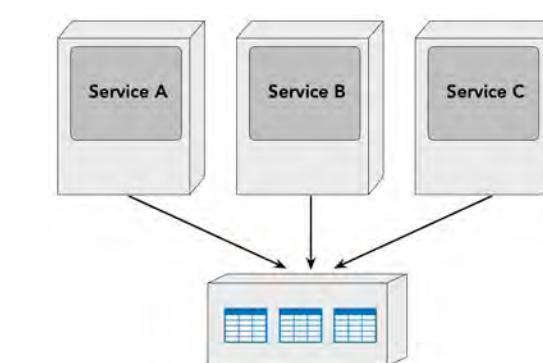
service  
types



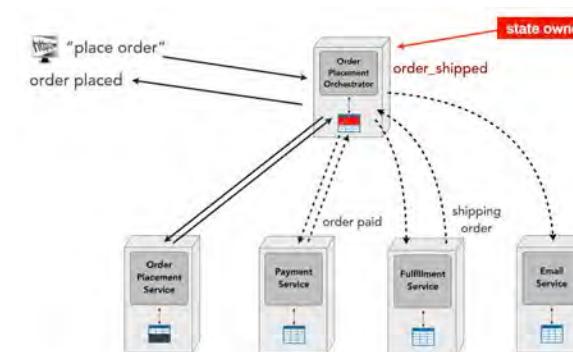
code sharing  
techniques



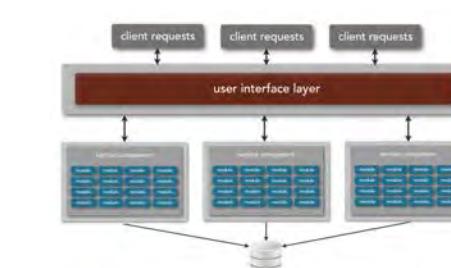
decomposing  
data



data ownership  
and access



managing  
workflows



hybrid  
architectures



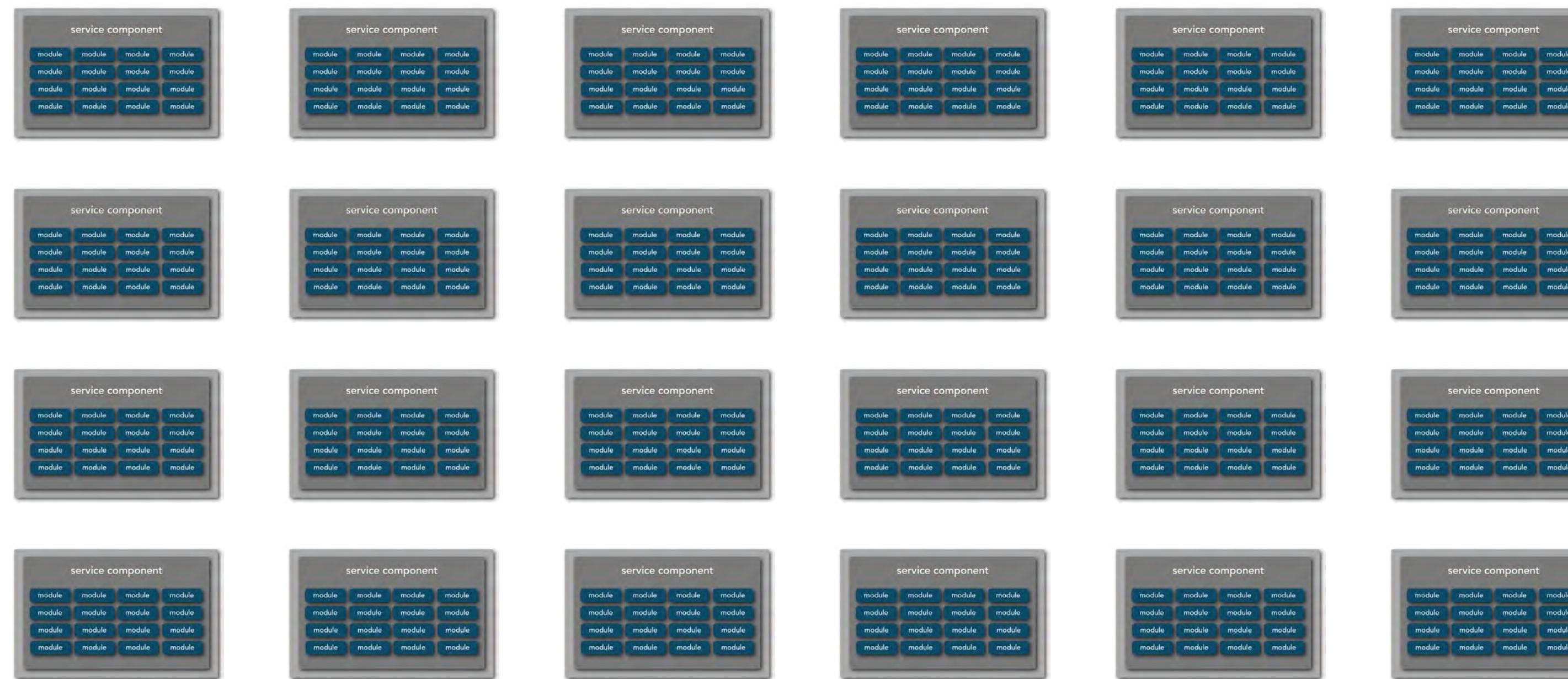
final words  
of advice

# Architectural Modularity

# architectural modularity



# architectural modularity





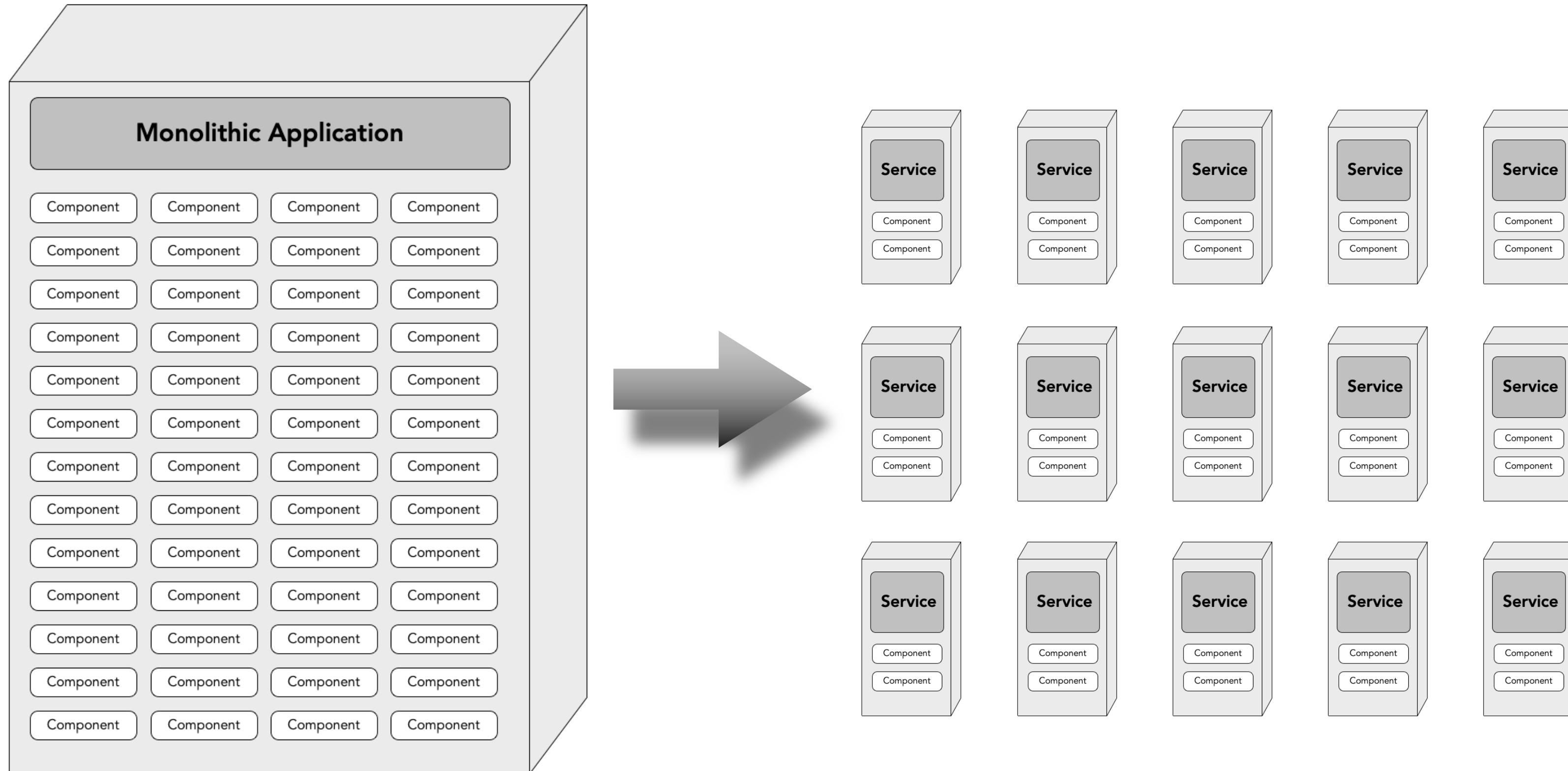




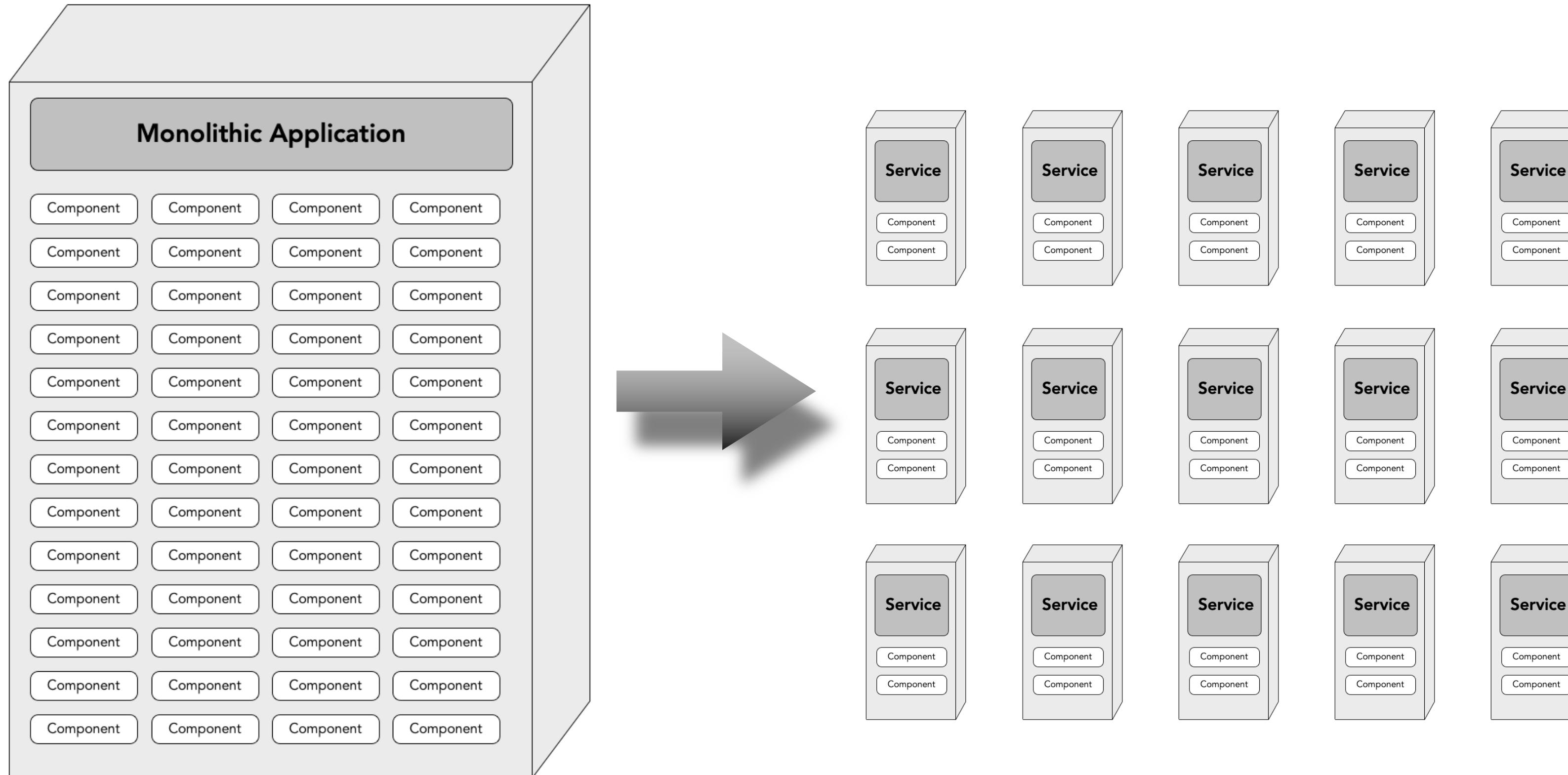




# architectural modularity

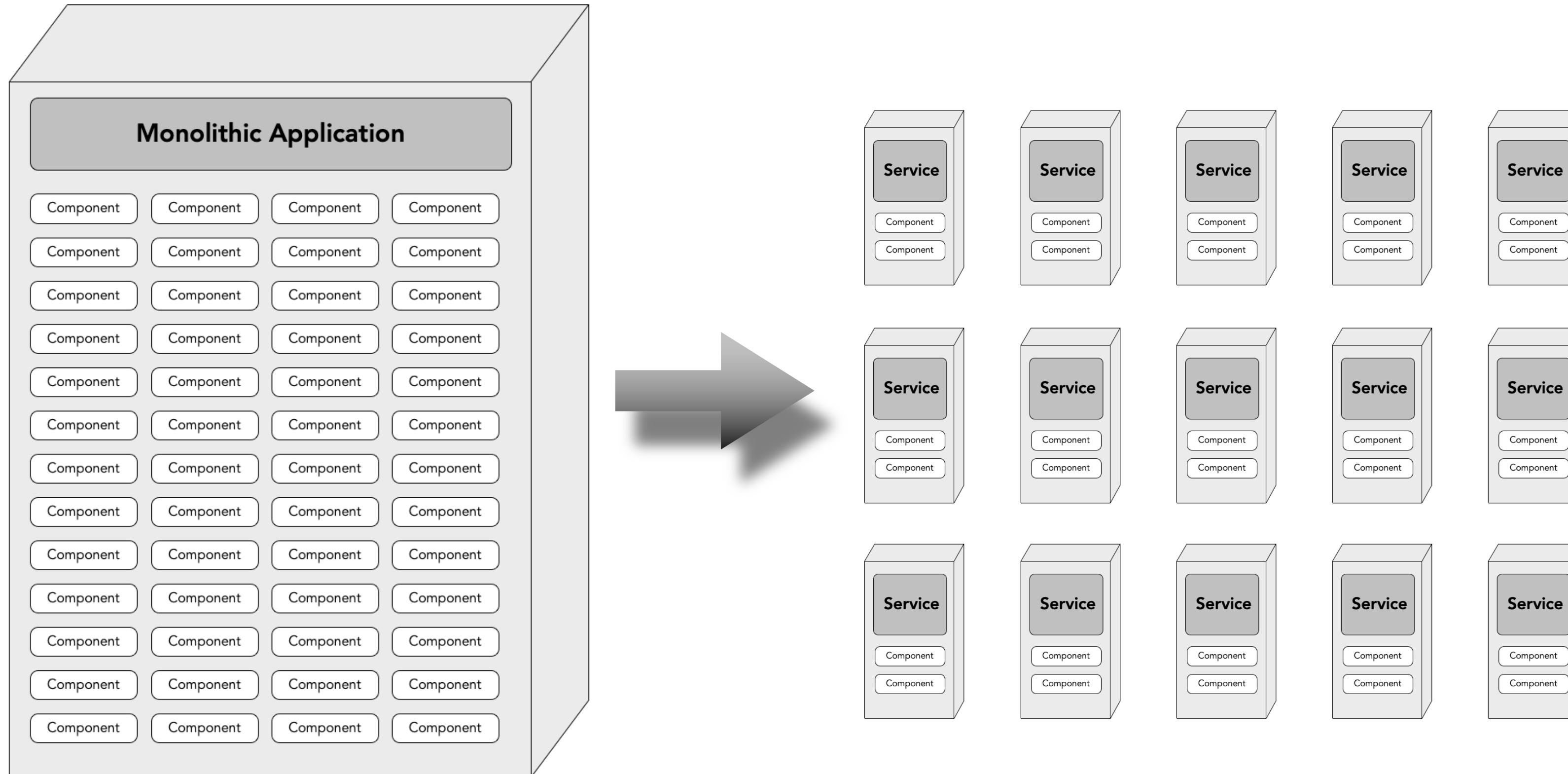


# architectural modularity



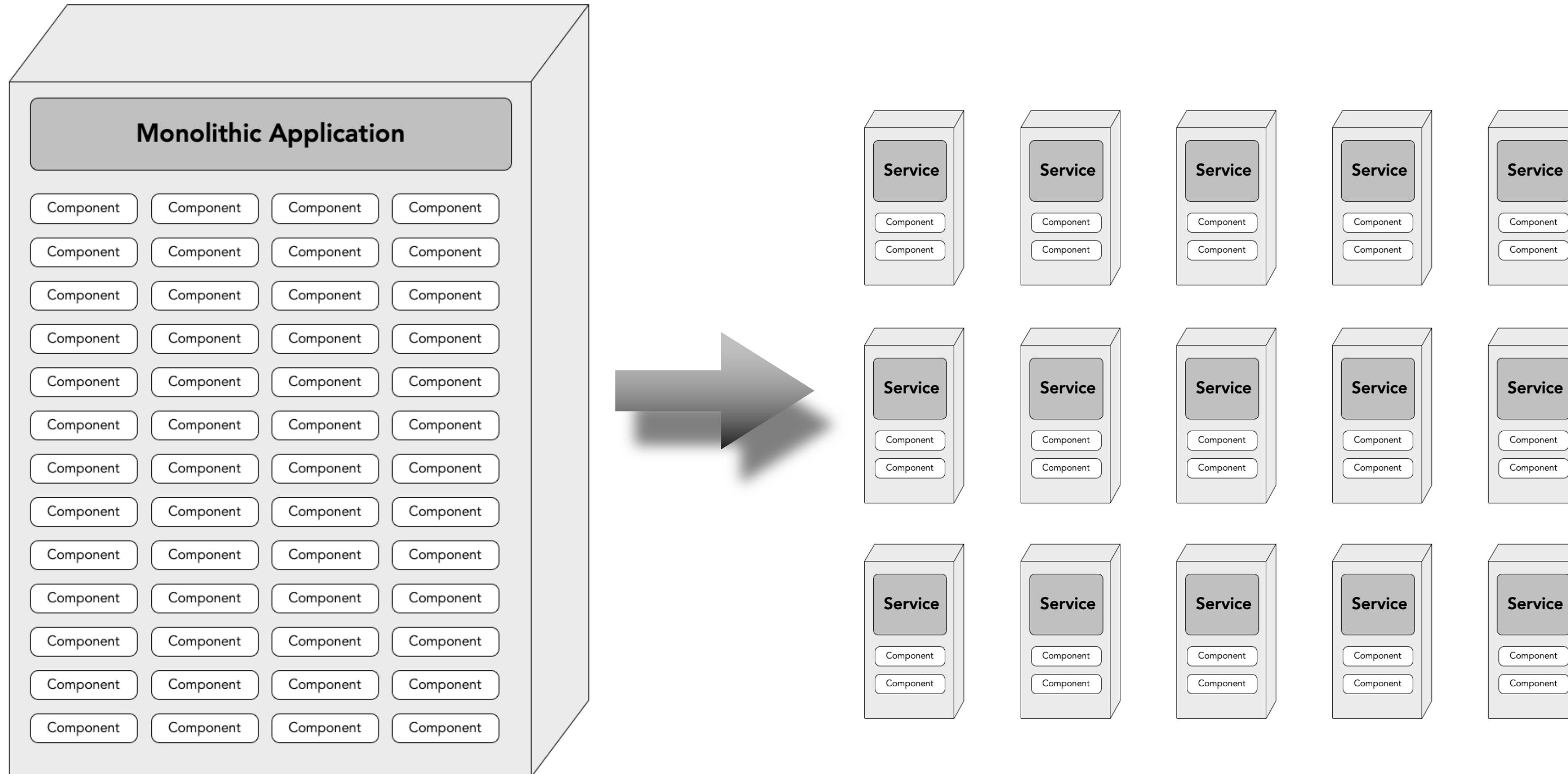
✓ maintainability

# architectural modularity



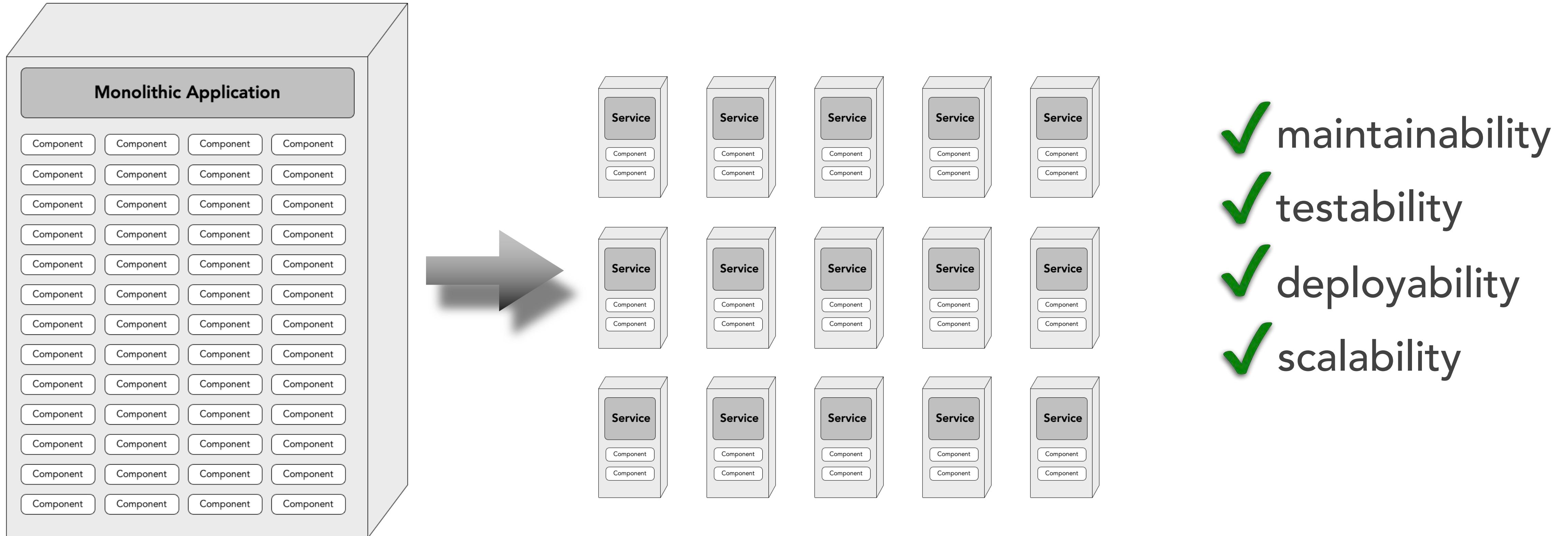
✓ maintainability  
✓ testability

# architectural modularity

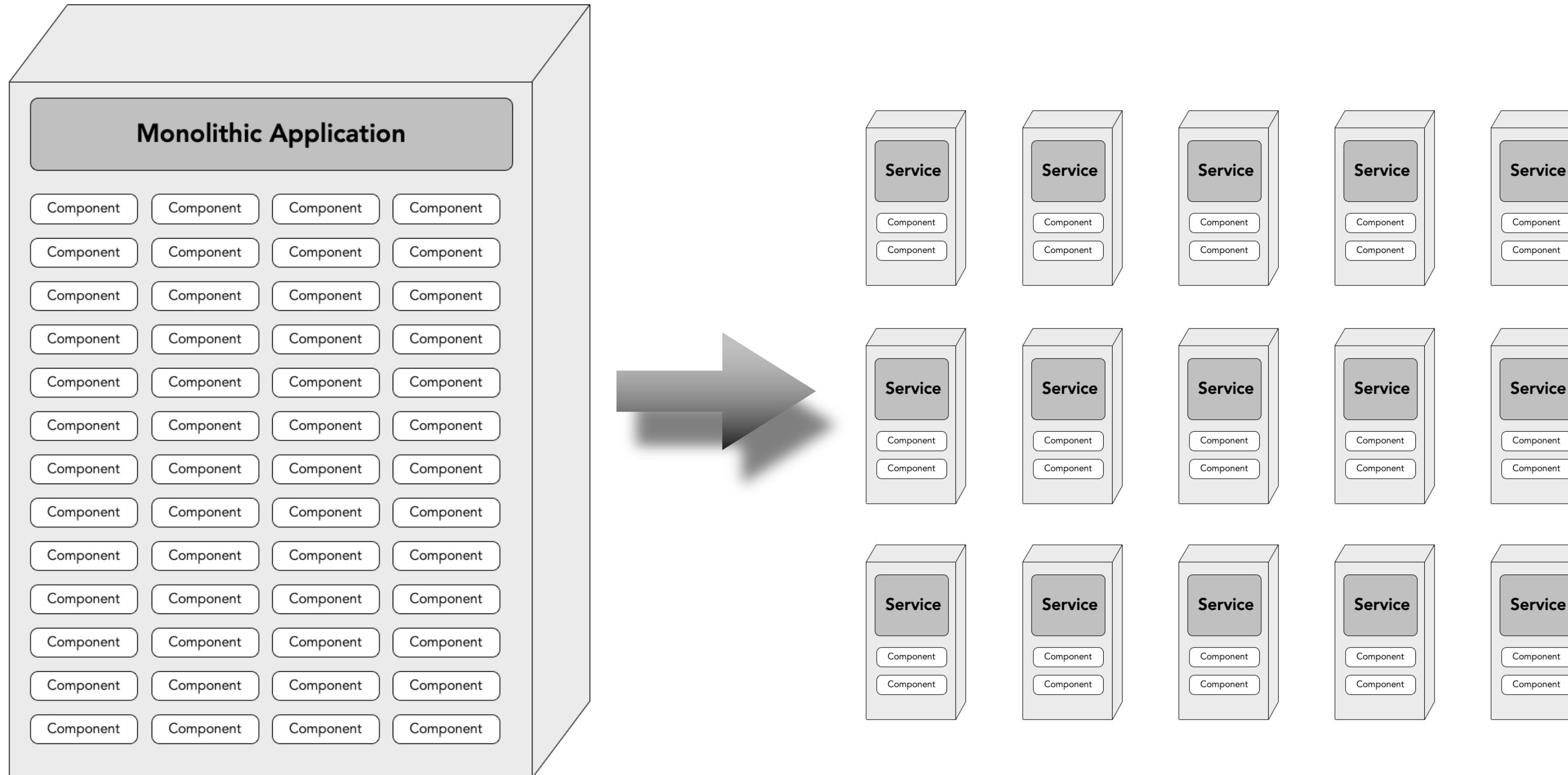


✓ maintainability  
✓ testability  
✓ deployability

# architectural modularity



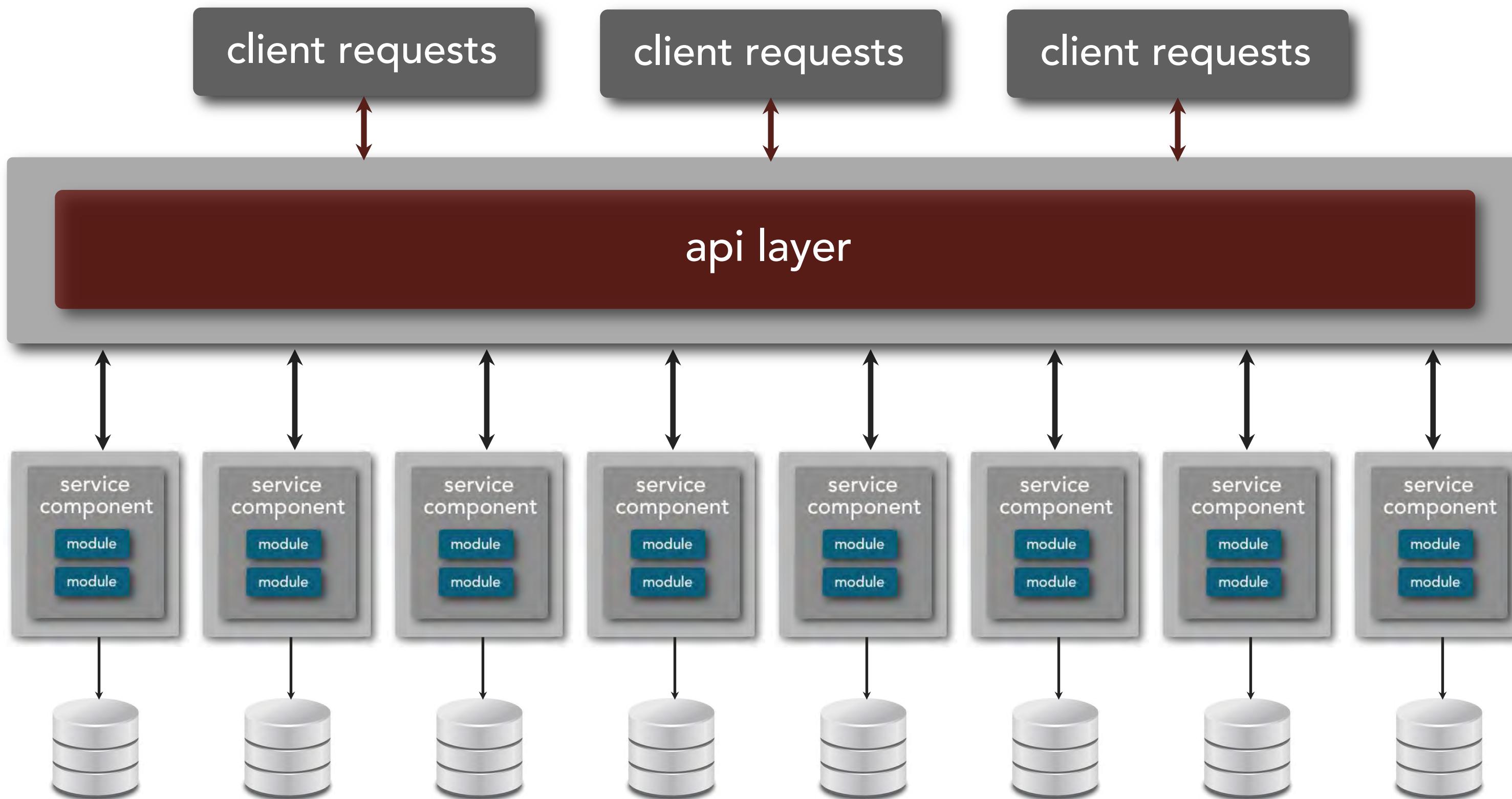
# architectural modularity



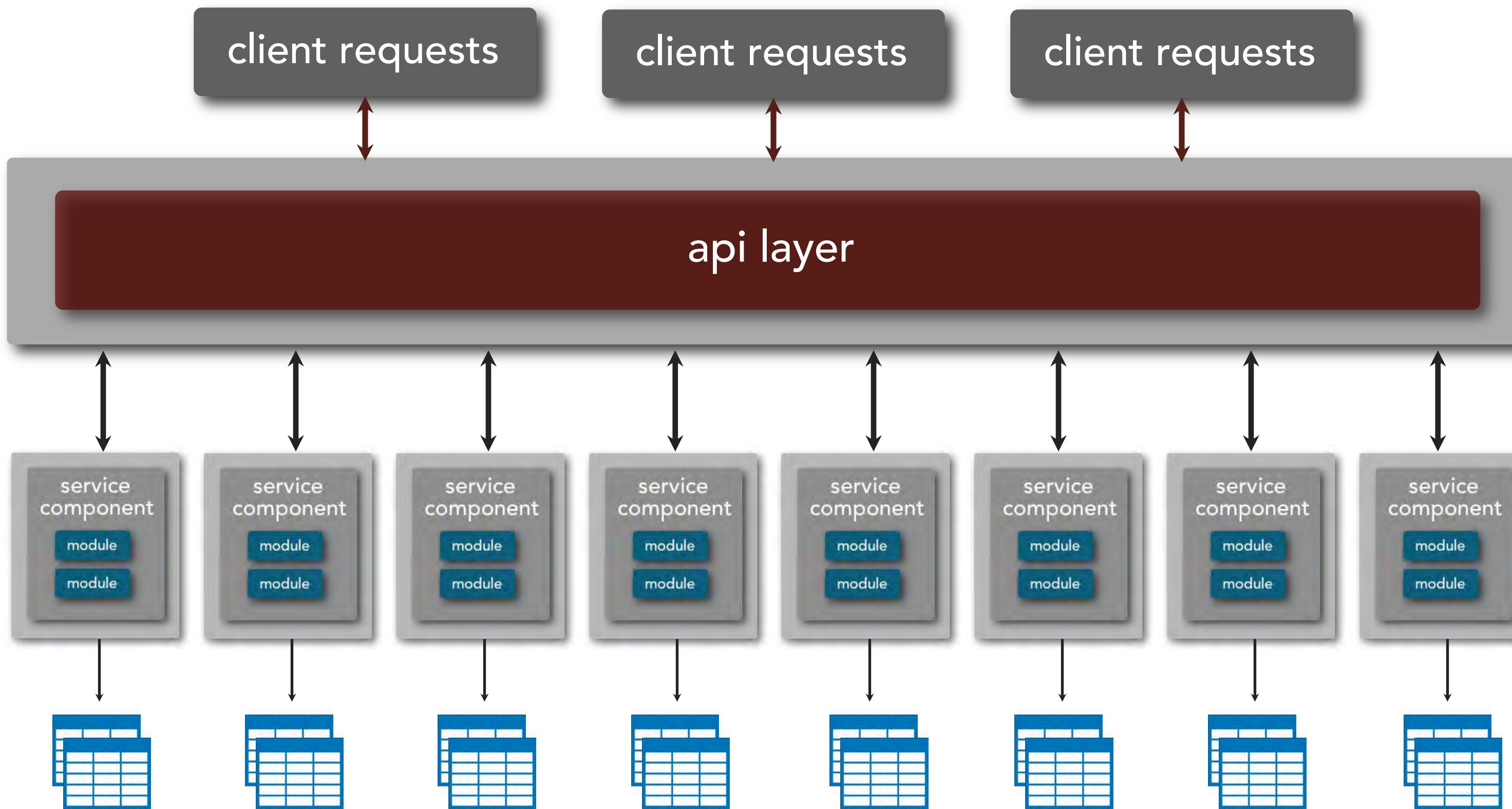
- ✓ maintainability
- ✓ testability
- ✓ deployability
- ✓ scalability
- ✓ availability

# Microservices Core Concepts

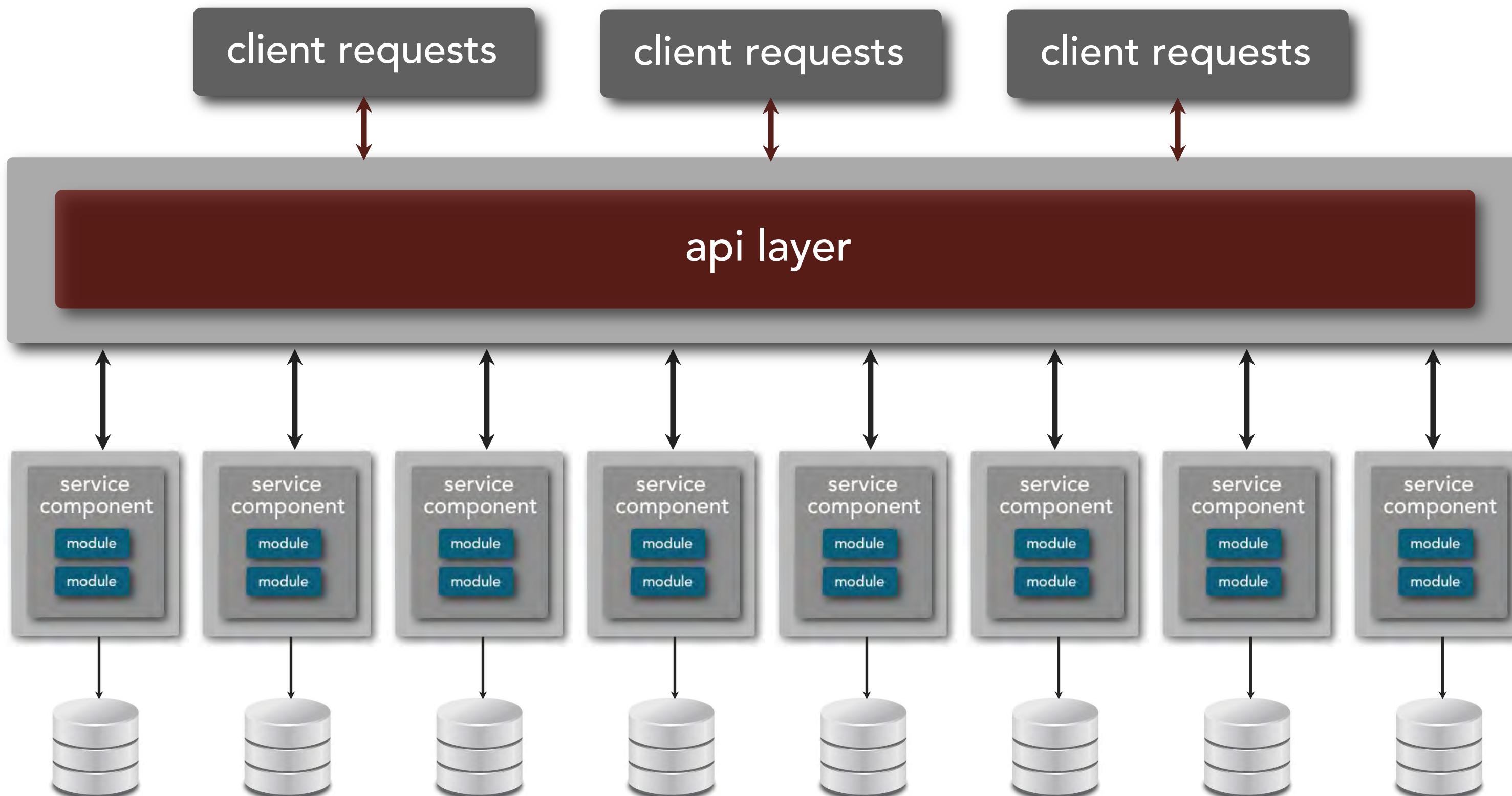
# microservices core concepts



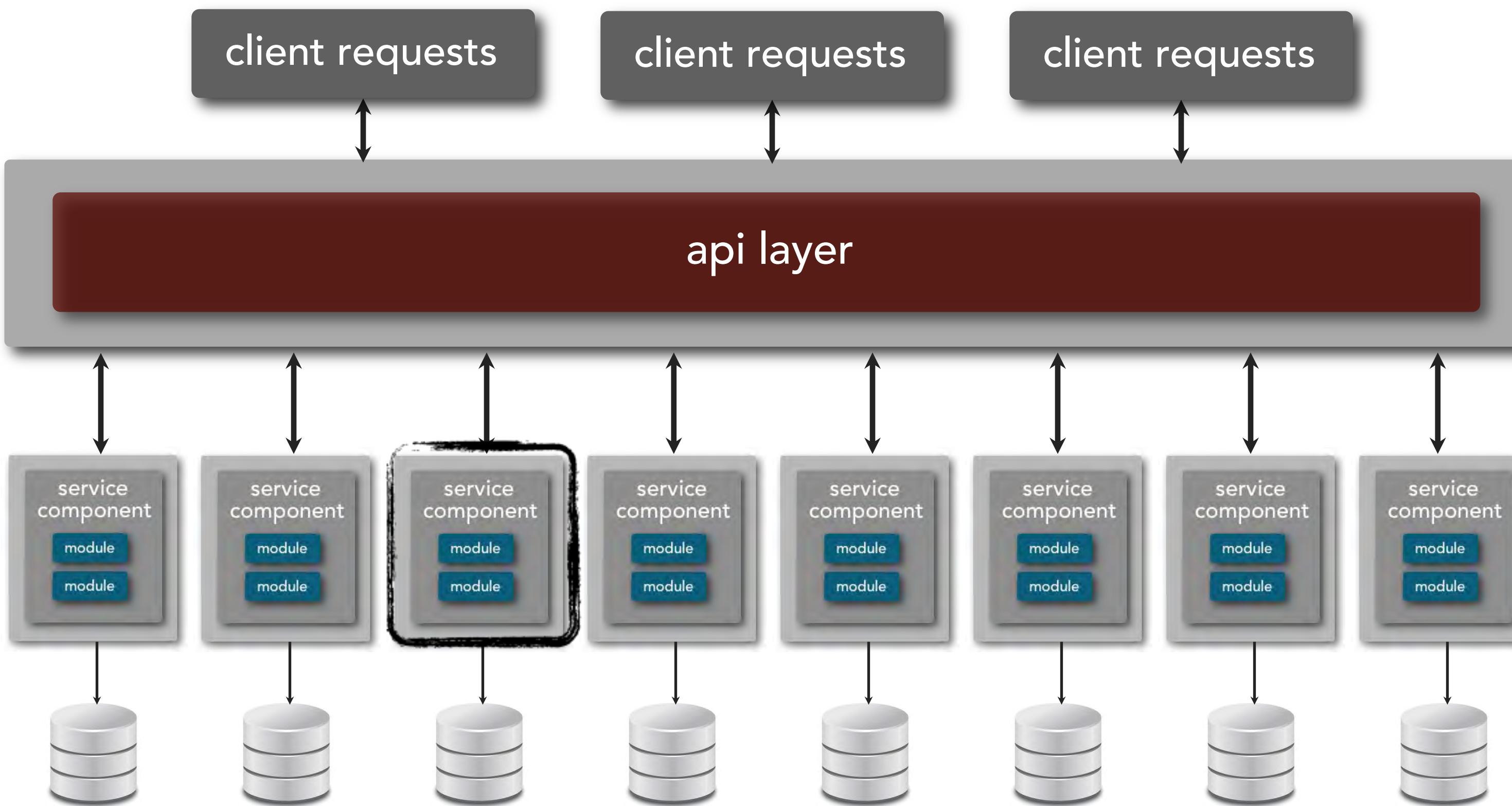
# microservices core concepts



# microservices core concepts

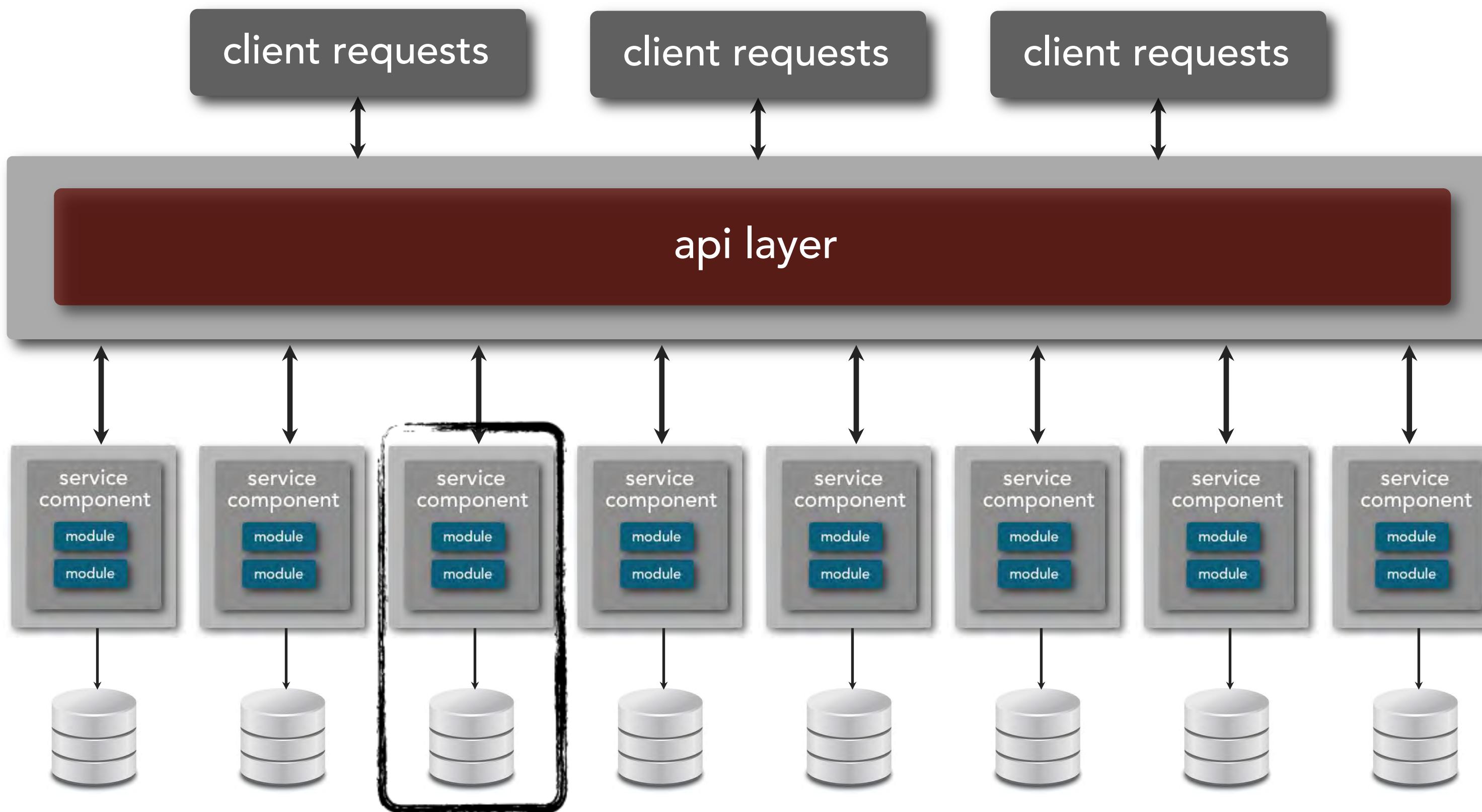


# microservices core concepts



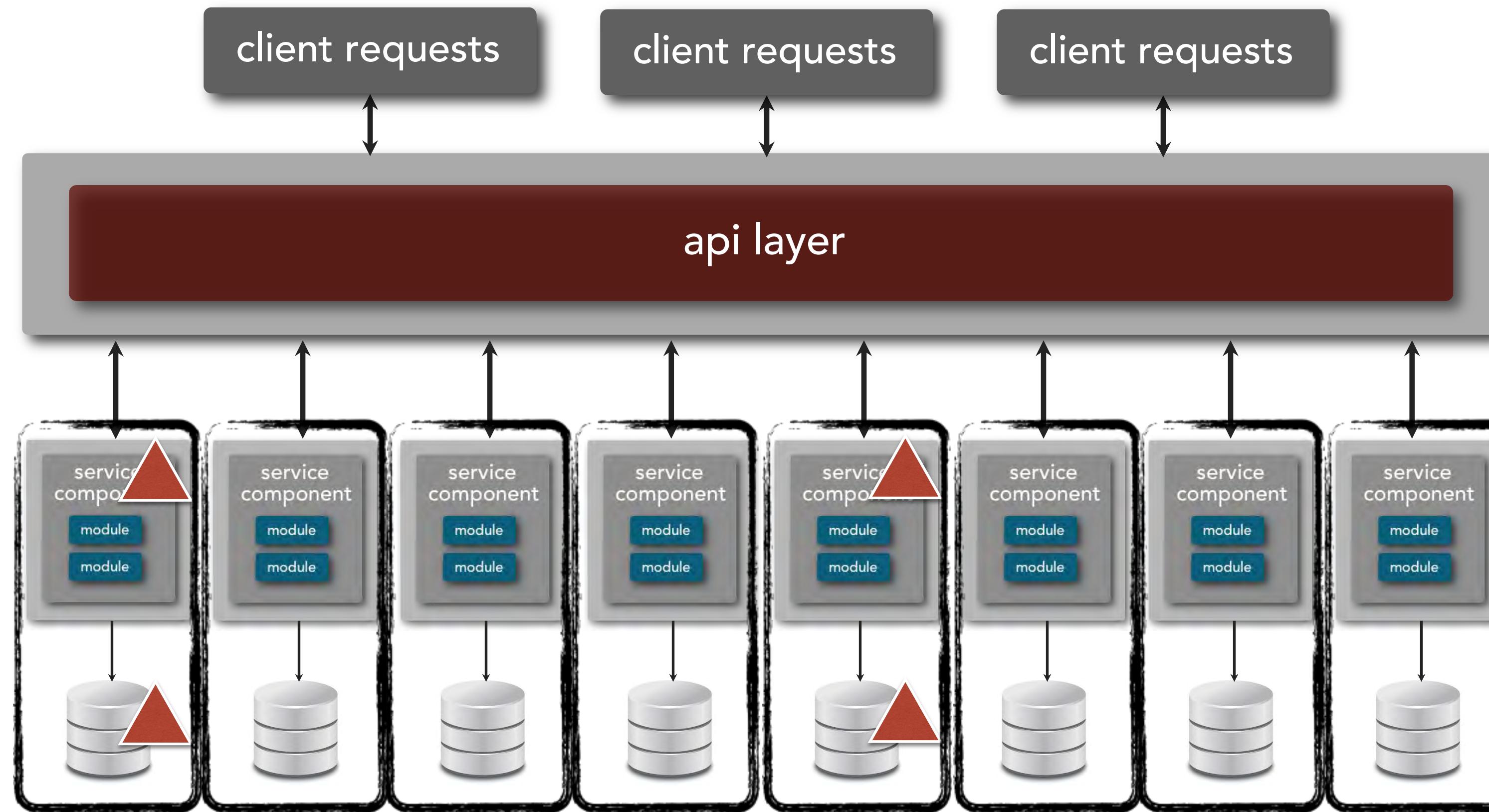
single-purpose, separately deployed unit of software  
that does *one thing* really well

# microservices core concepts



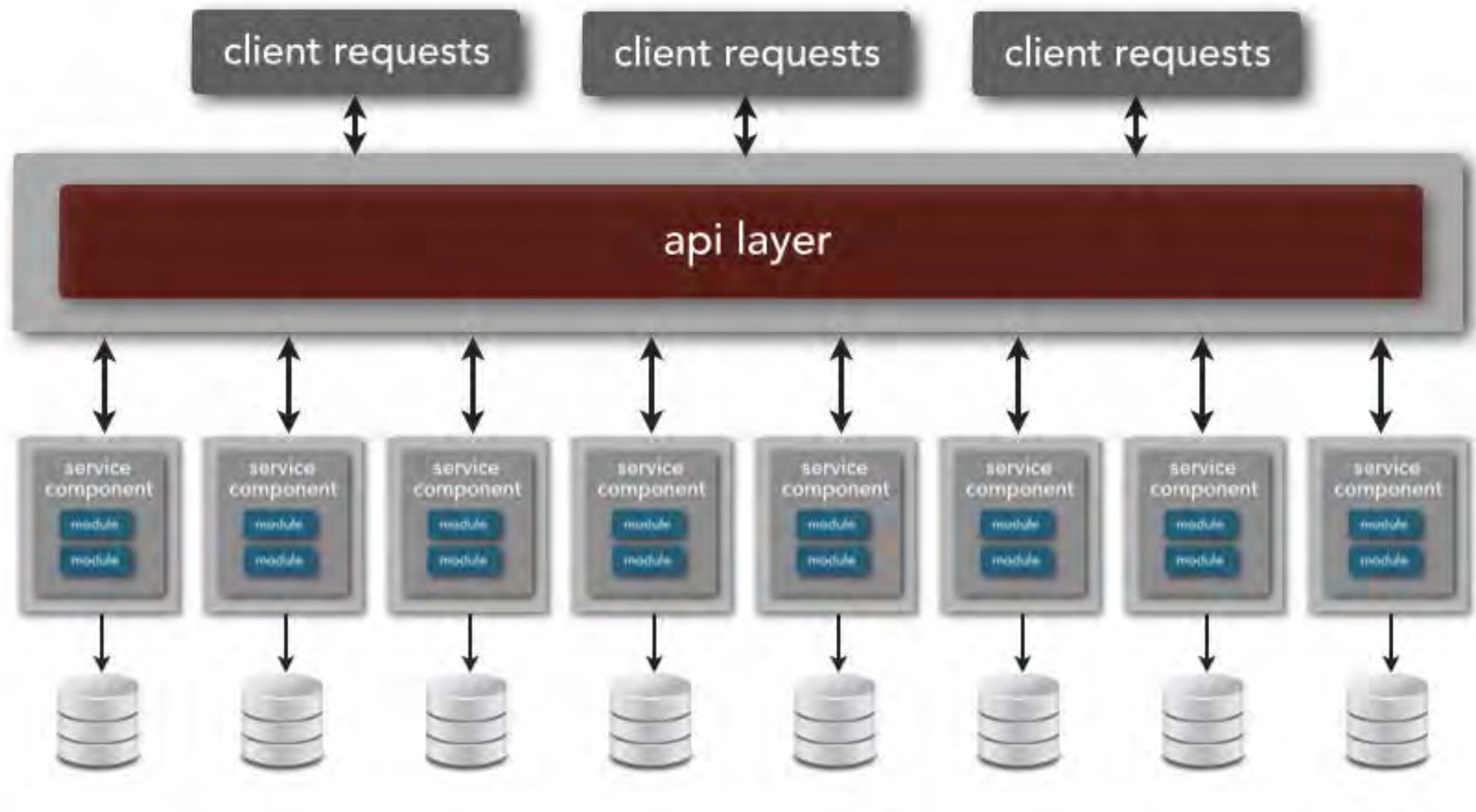
bounded context (share nothing architecture)

# microservices core concepts

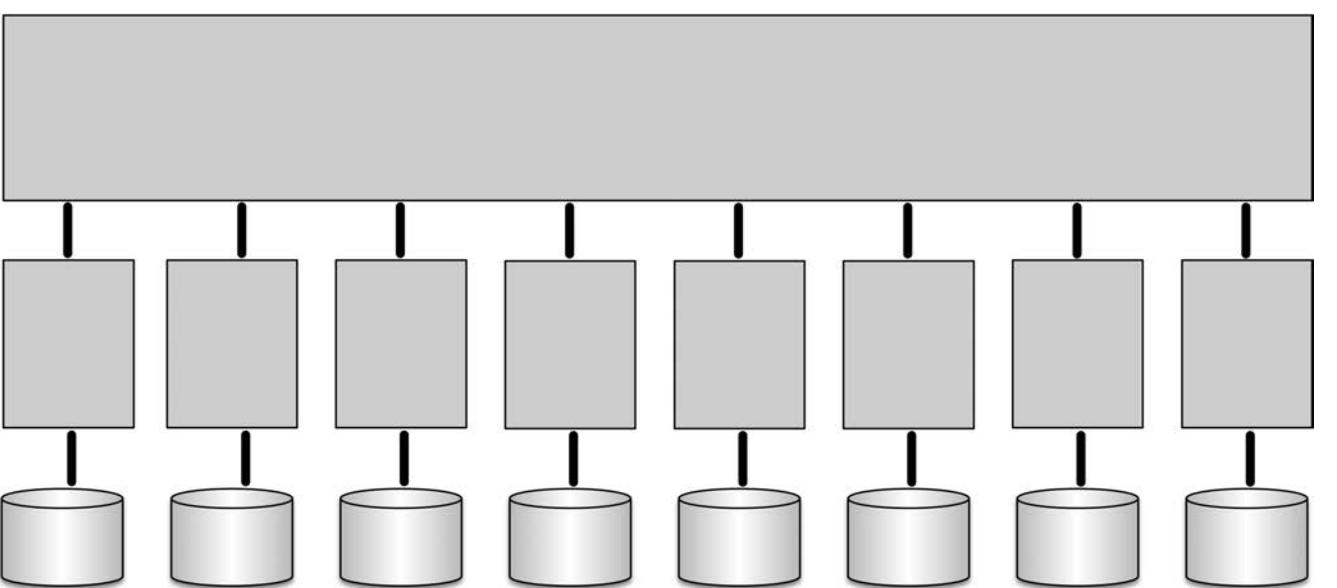


bounded context (share nothing architecture)

# microservices core concepts



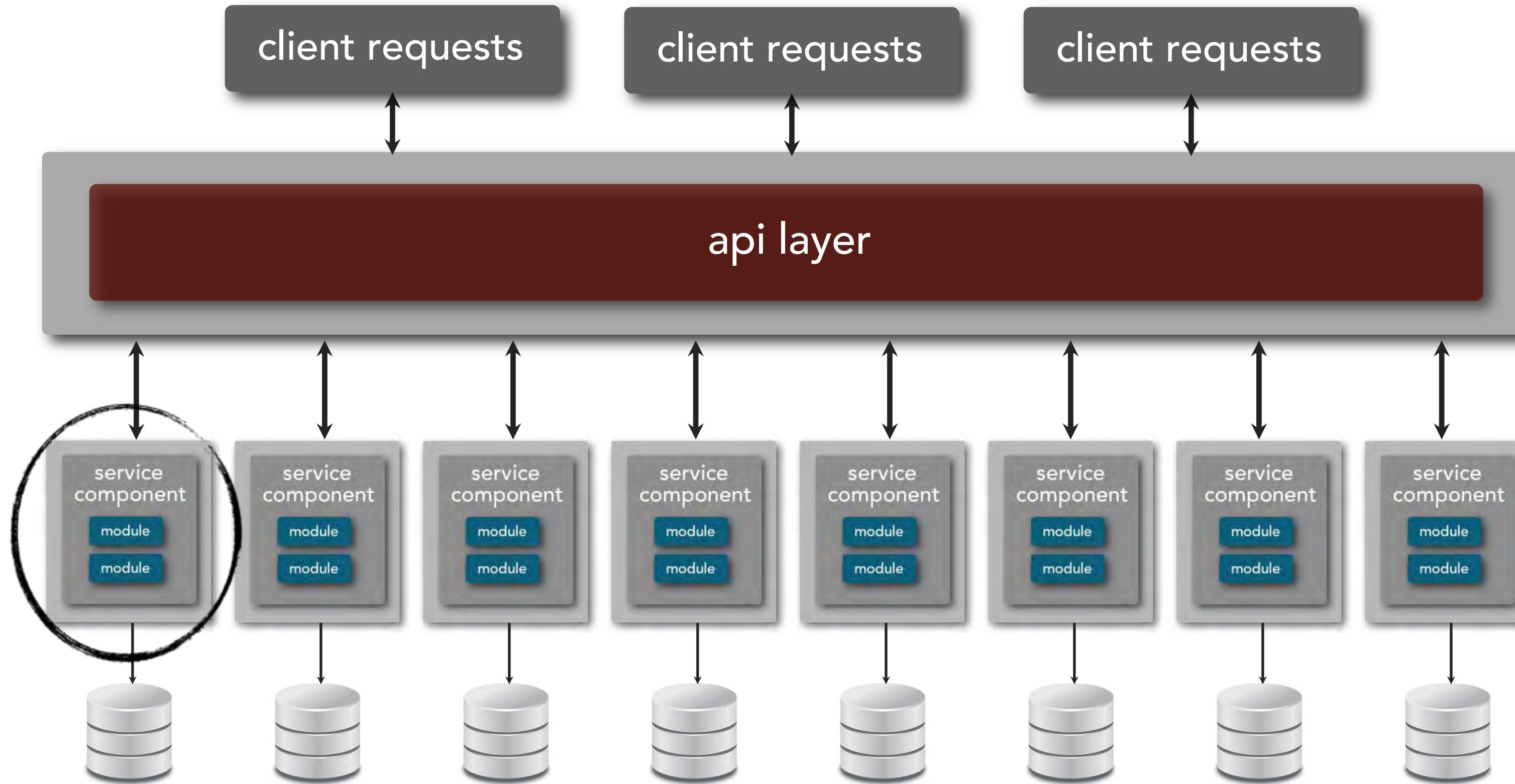
single purpose functions  
deployed as separate units



maintainability	★★★★★
deployability	★★★★★
testability	★★★★★
fault-tolerance	★★★★★
scalability	★★★★★
evolvability	★★★★★
performance	★★
simplicity	★
cost	★
workflow	★

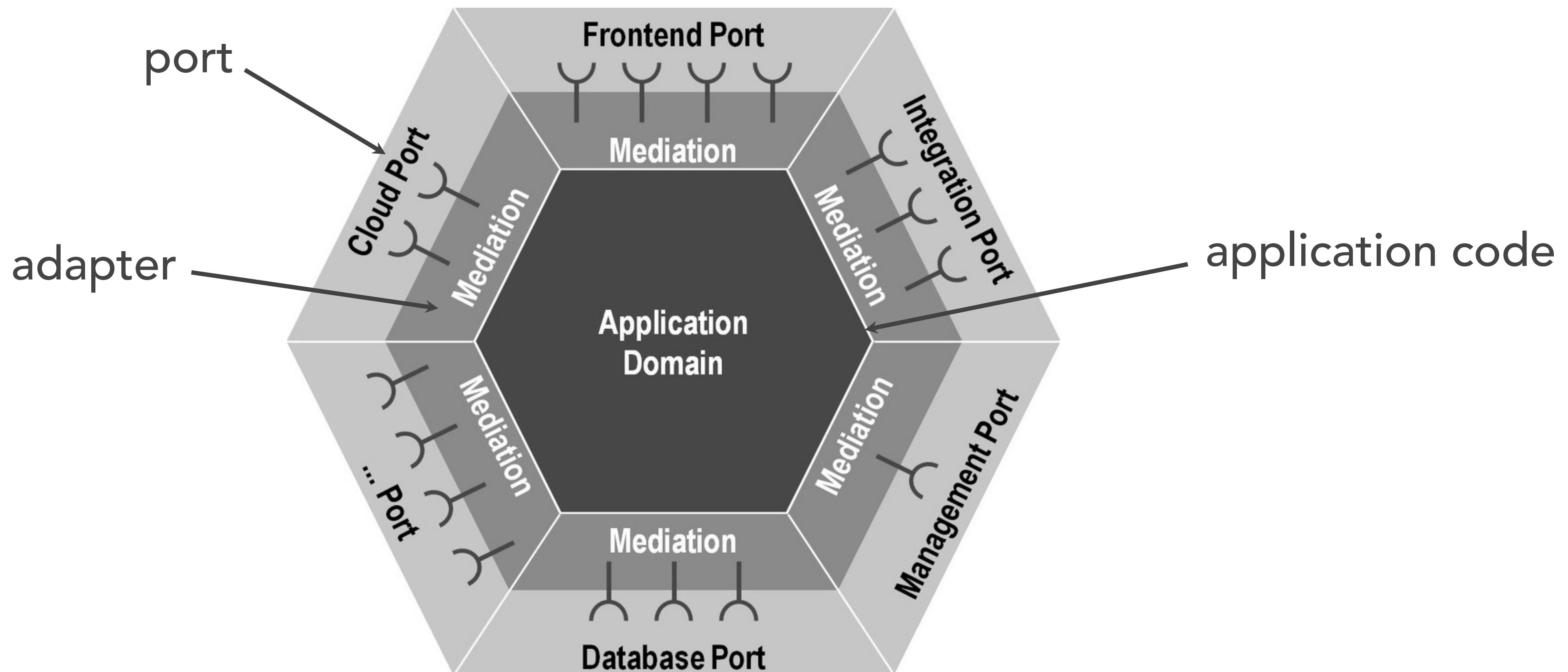
# Service Design

# service design



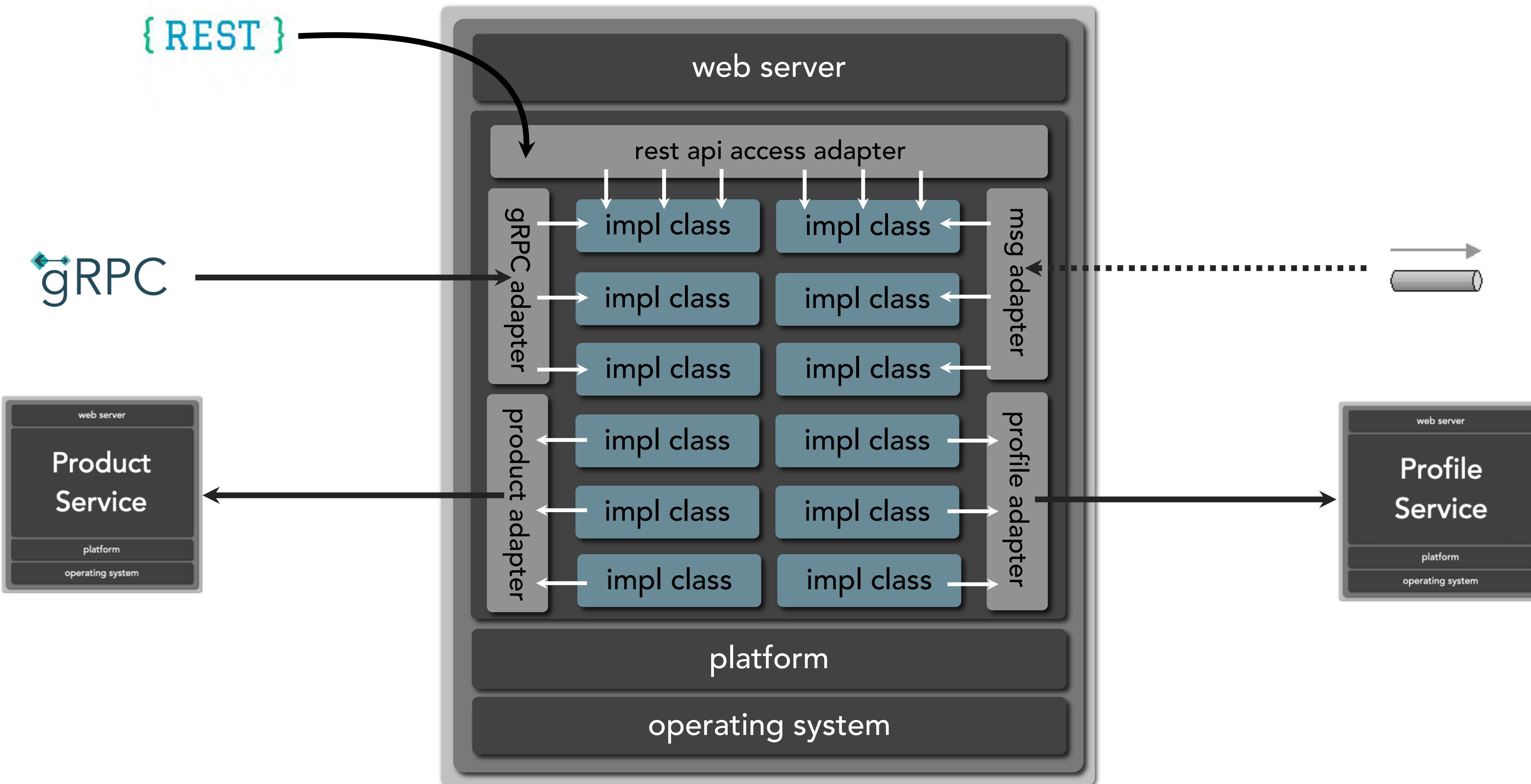
# service design

*micro-hexagonal design to support protocol agnostic processing*



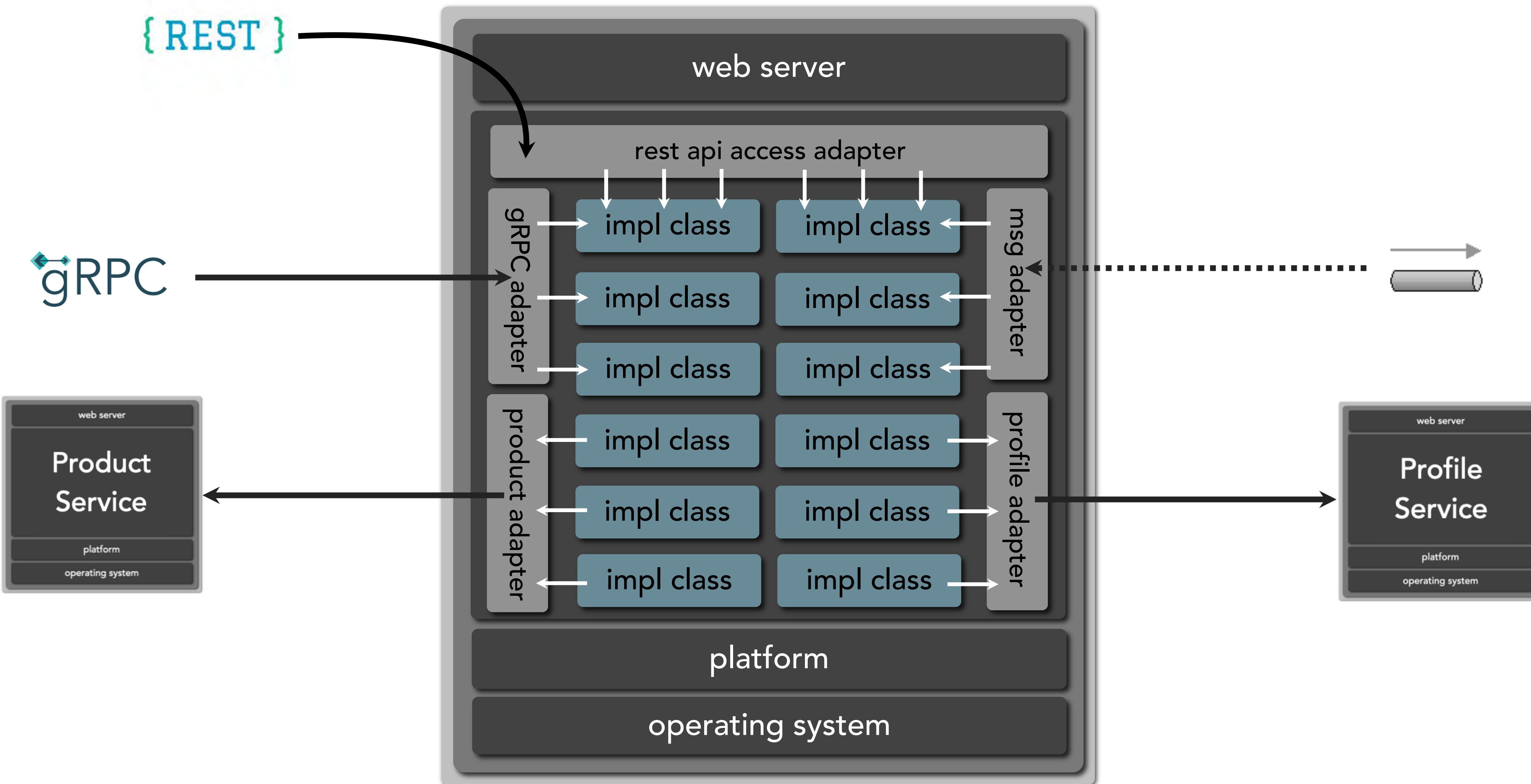
# service design

*micro-hexagonal design to support protocol agnostic processing*



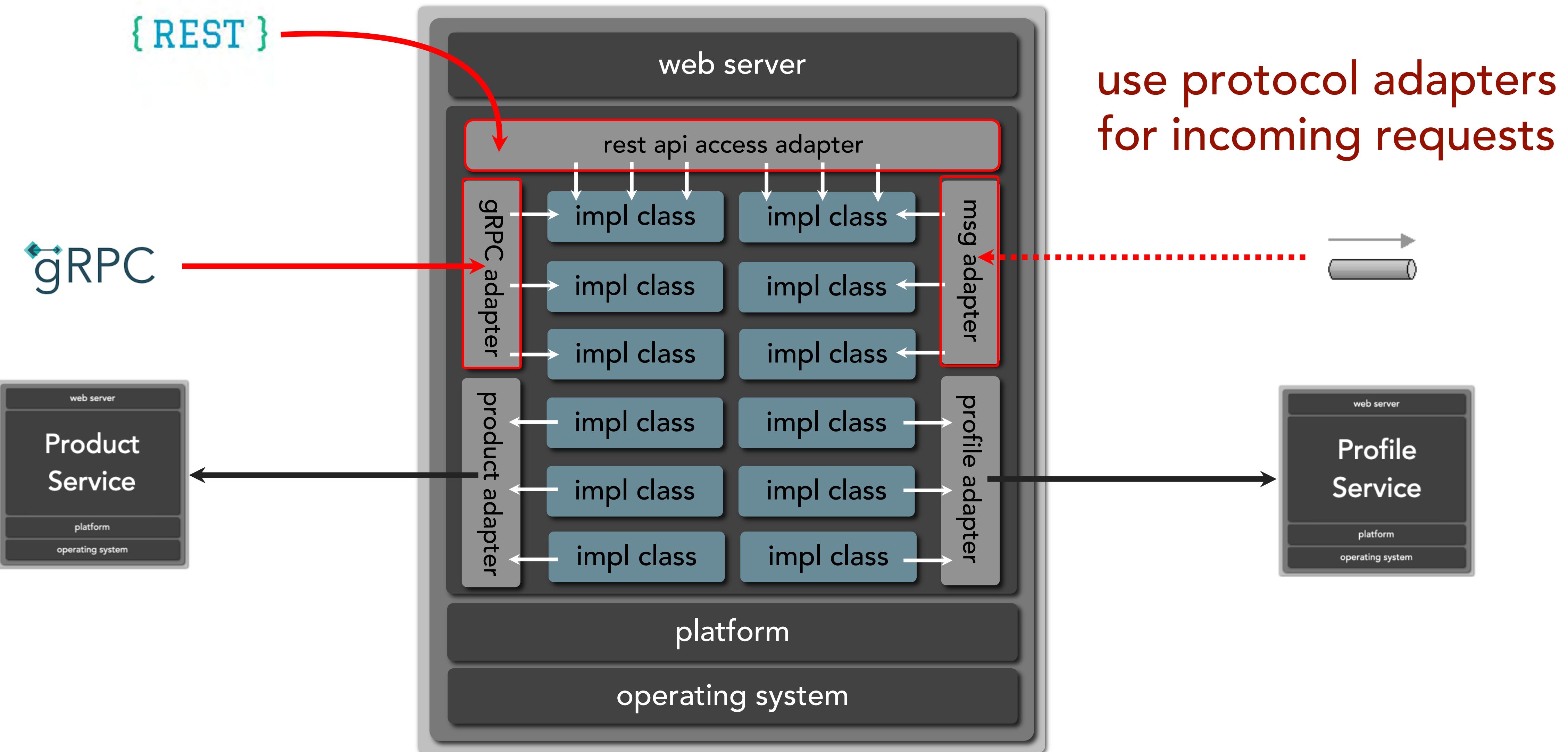
# service design

*micro-hexagonal design to support protocol agnostic processing*



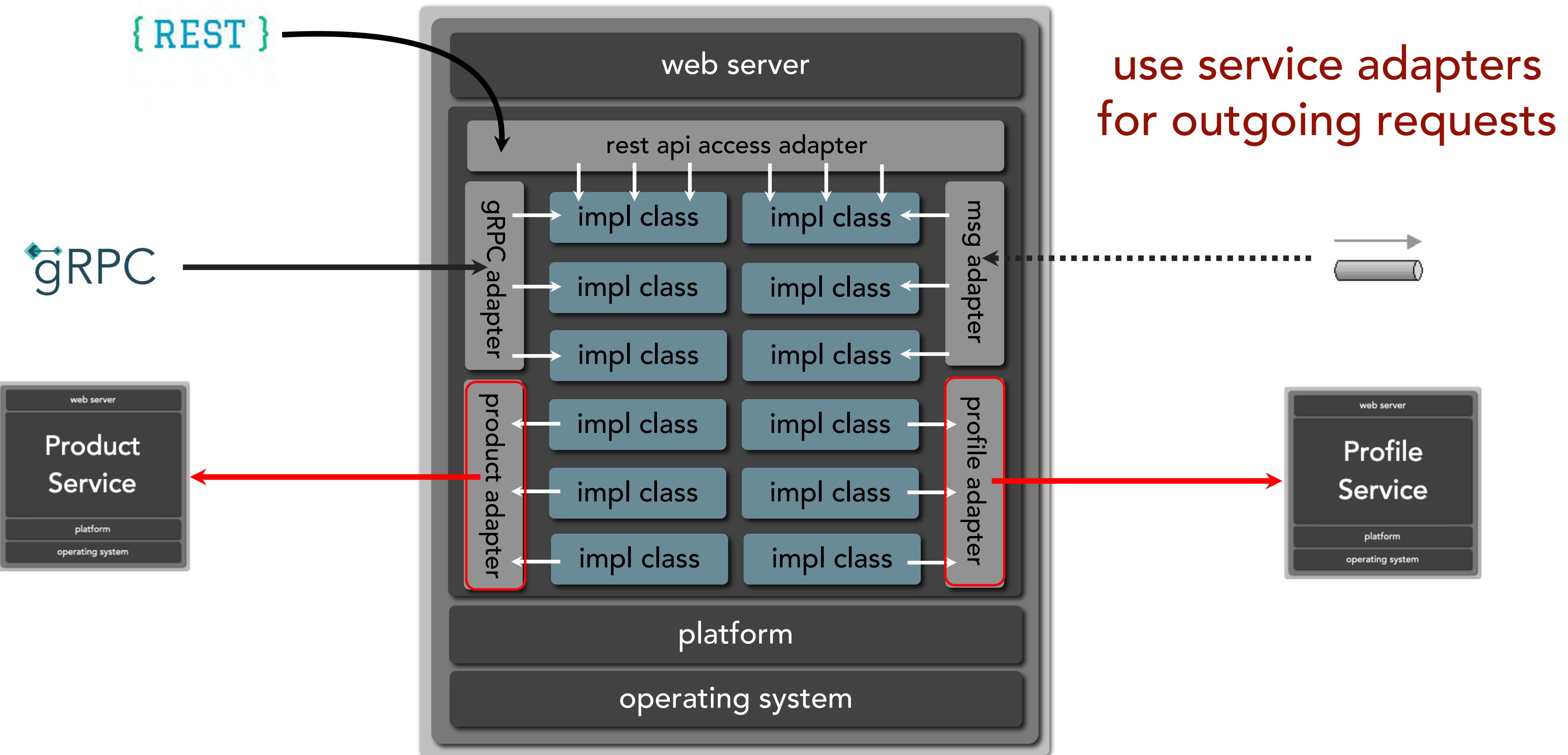
# service design

micro-hexagonal design to support protocol agnostic processing



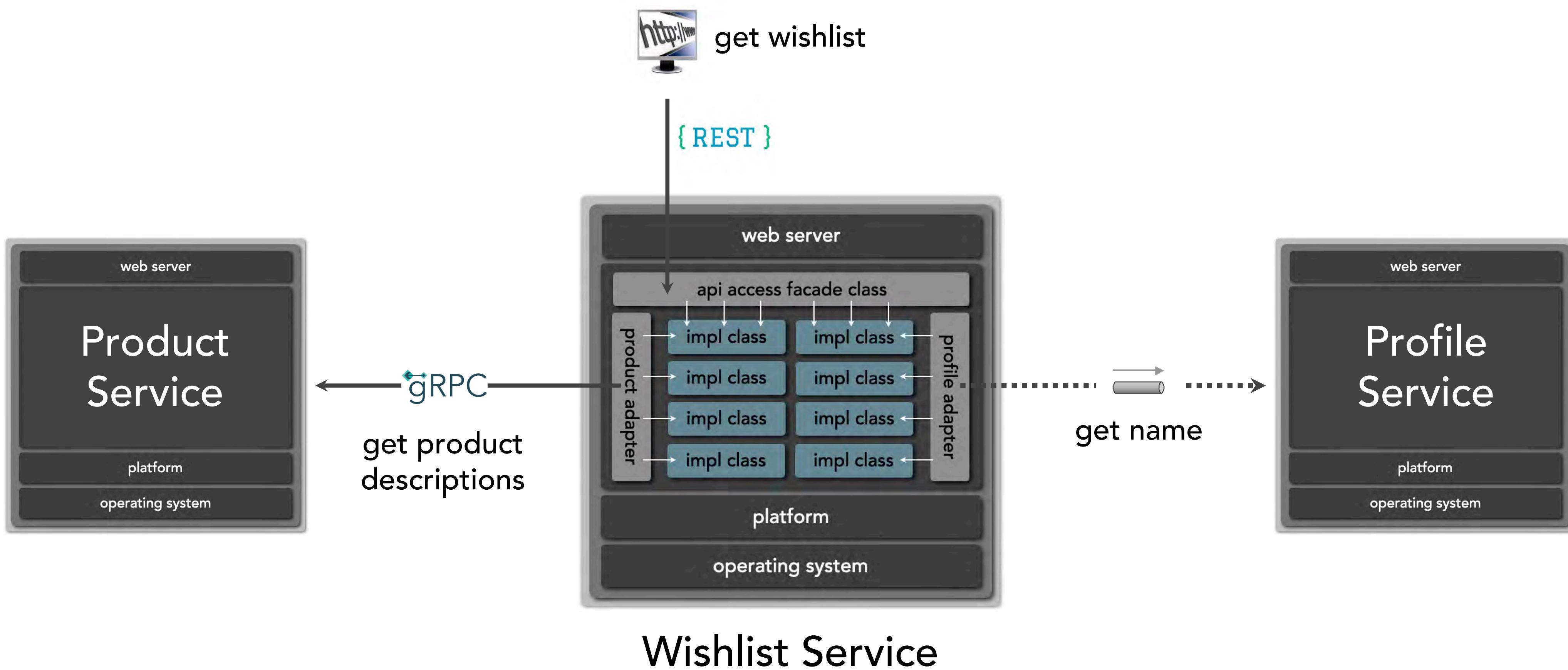
# service design

micro-hexagonal design to support protocol agnostic processing



# service design

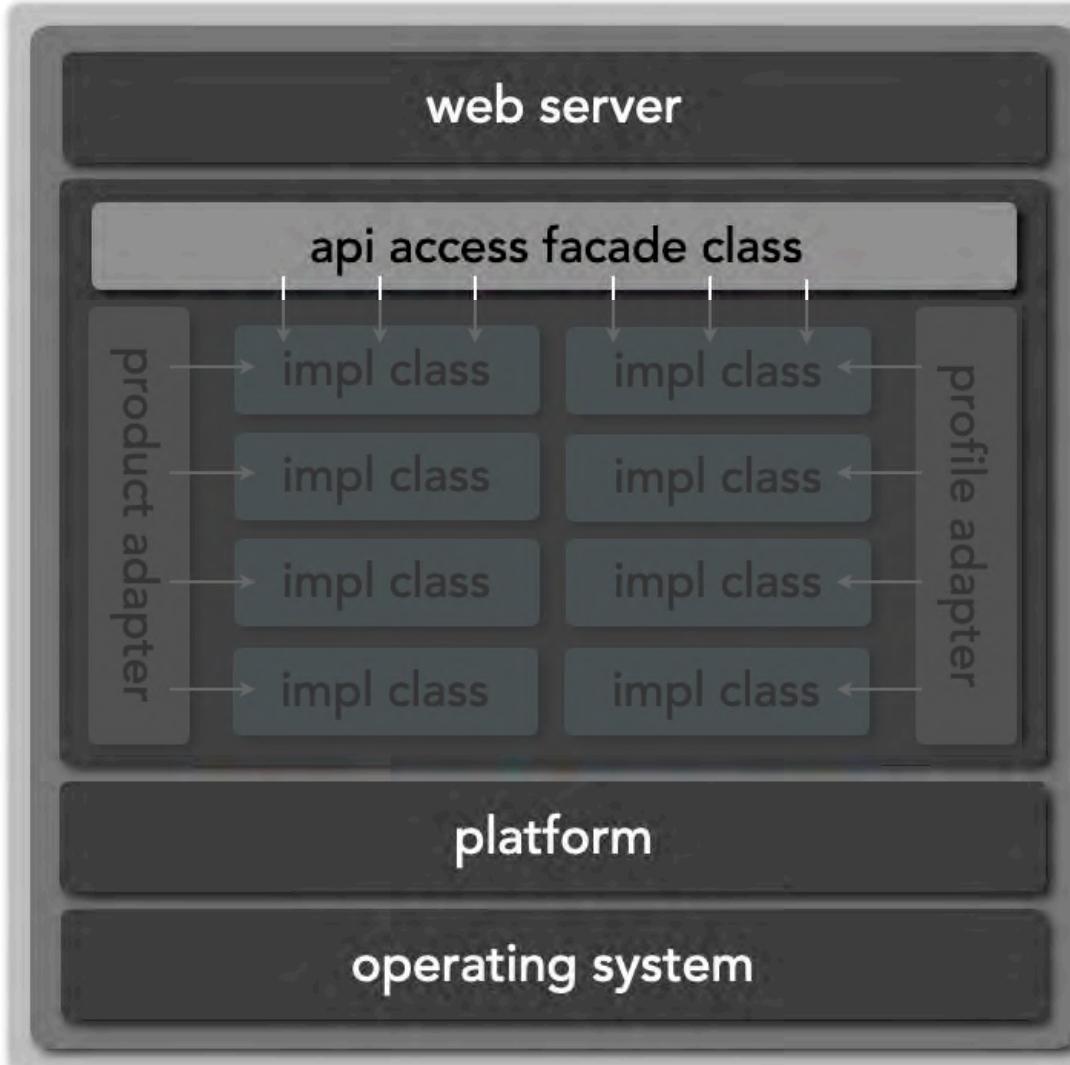
*micro-hexagonal design to support protocol agnostic processing*



# service design

*micro-hexagonal design to support protocol agnostic processing*

## API REST Access Adapter Class

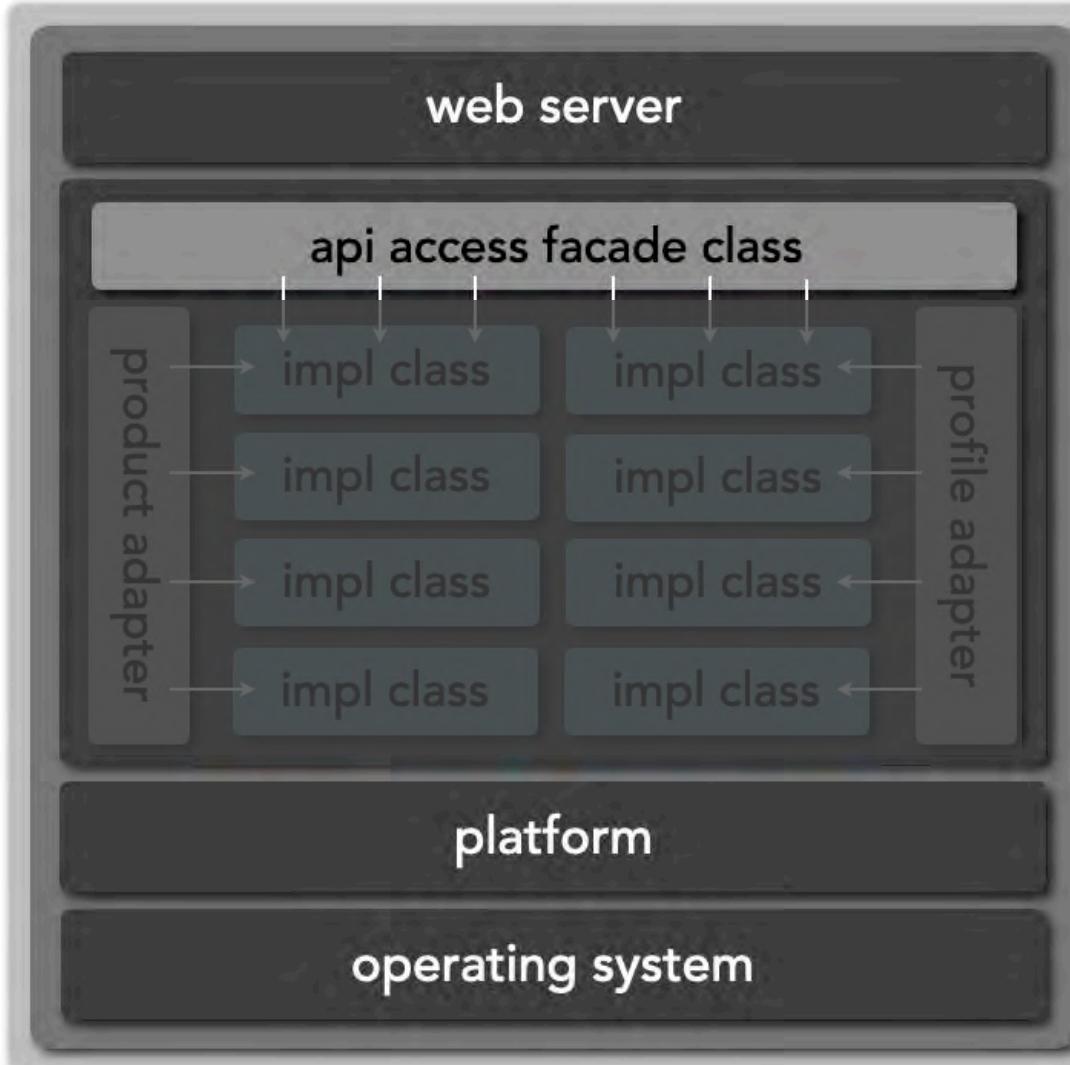


```
@Path( "/wishlist" )
@Produces(MediaType.APPLICATION_JSON)
@GET
public String getwishlist(long customerId) {
    WishlistImpl wishlist = new WishlistImpl();
    WishlistItems wishlistItems =
        wishlist.getWishlistItems(customerId);
    return convertToJson(wishlistItems);
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## API REST Access Adapter Class

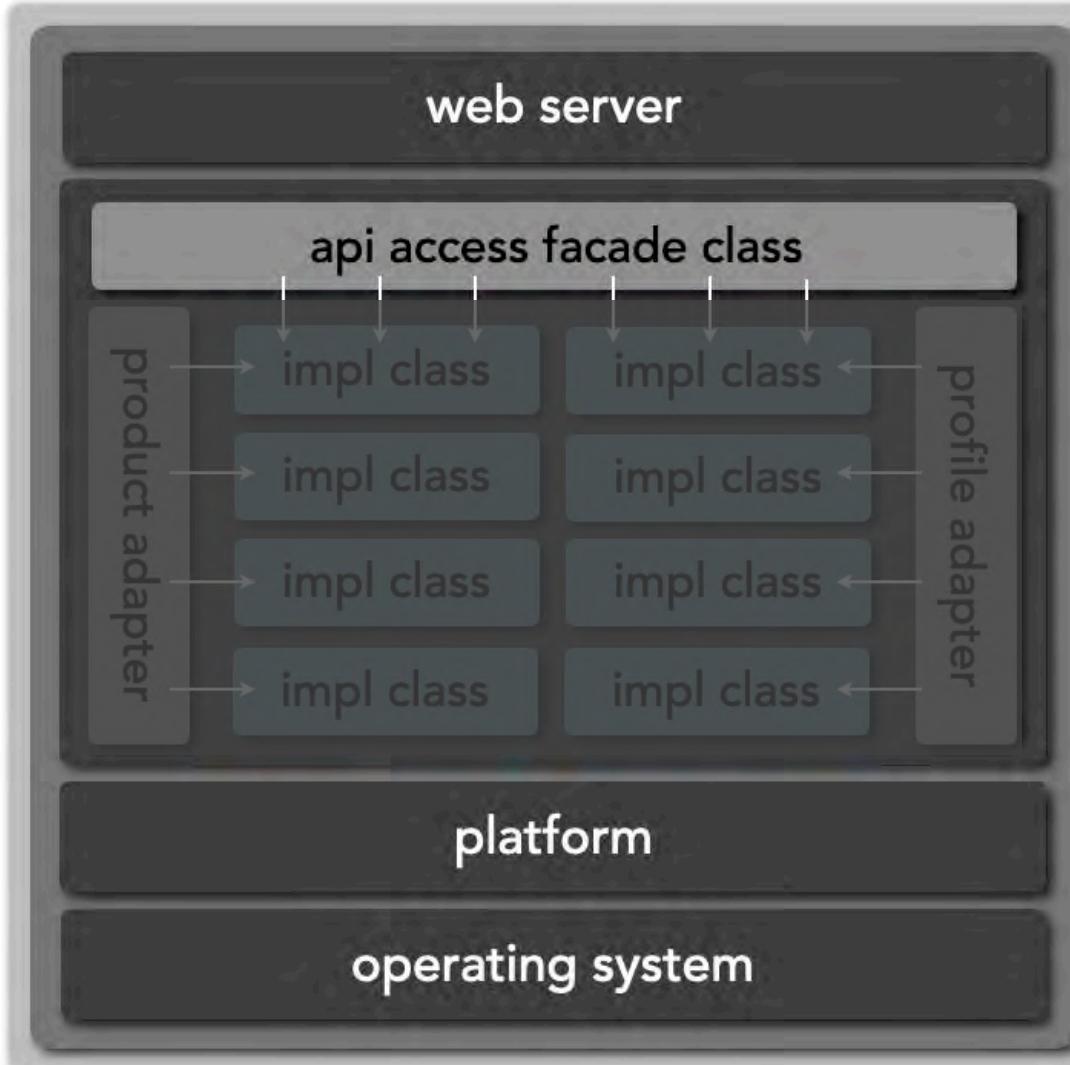


```
@Path( "/wishlist" )
@Produces(MediaType.APPLICATION_JSON)
@GET
public String getwishlist(long customerId) {
    WishlistImpl wishlist = new WishlistImpl();
    WishlistItems wishlistItems =
        wishlist.getwishlistItems(customerId);
    return convertToJson(wishlistItems);
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## API REST Access Adapter Class

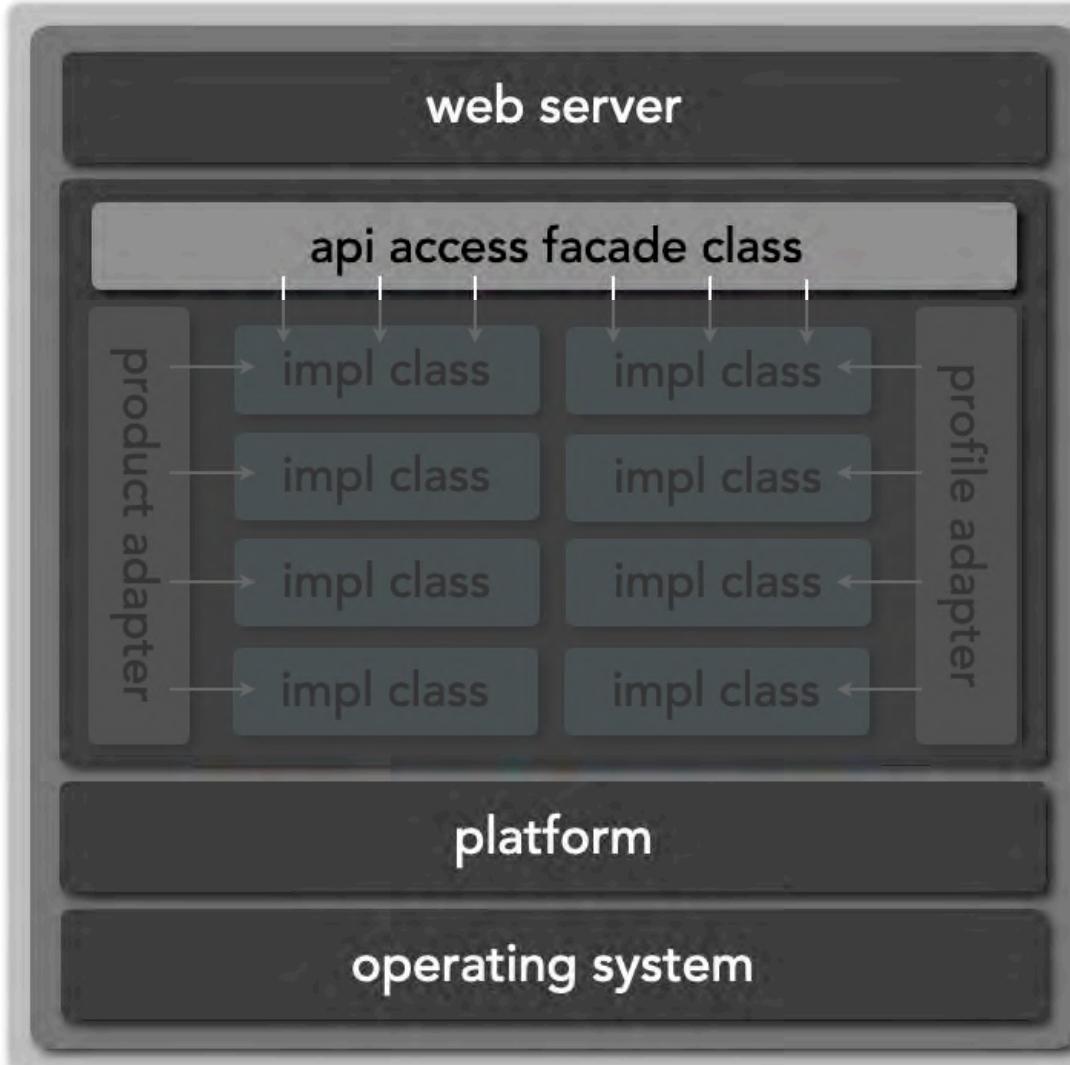


```
@Path( "/wishlist" )
@Produces(MediaType.APPLICATION_JSON)
@GET
public String getwishlist(long customerId) {
    WishlistImpl wishlist = new WishlistImpl();
    WishlistItems wishlistItems =
        wishlist.getwishlistItems(customerId);
    return convertToJson(wishlistItems);
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## API REST Access Adapter Class

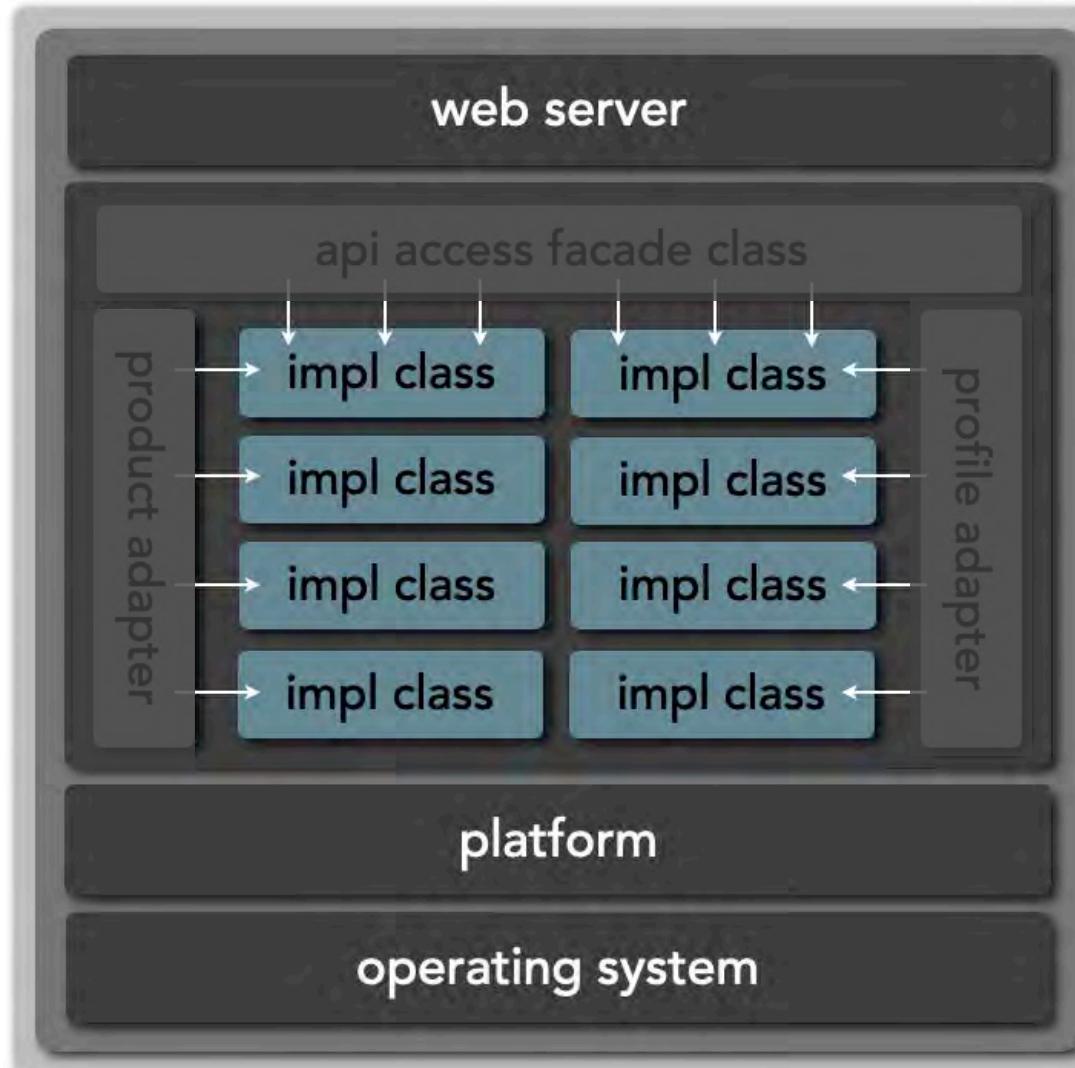


```
@Path( "/wishlist" )
@Produces(MediaType.APPLICATION_JSON)
@GET
public String getwishlist(long customerId) {
    WishlistImpl wishlist = new WishlistImpl();
    WishlistItems wishlistItems =
        wishlist.getWishlistItems(customerId);
    return convertToJson(wishlistItems);
}
```

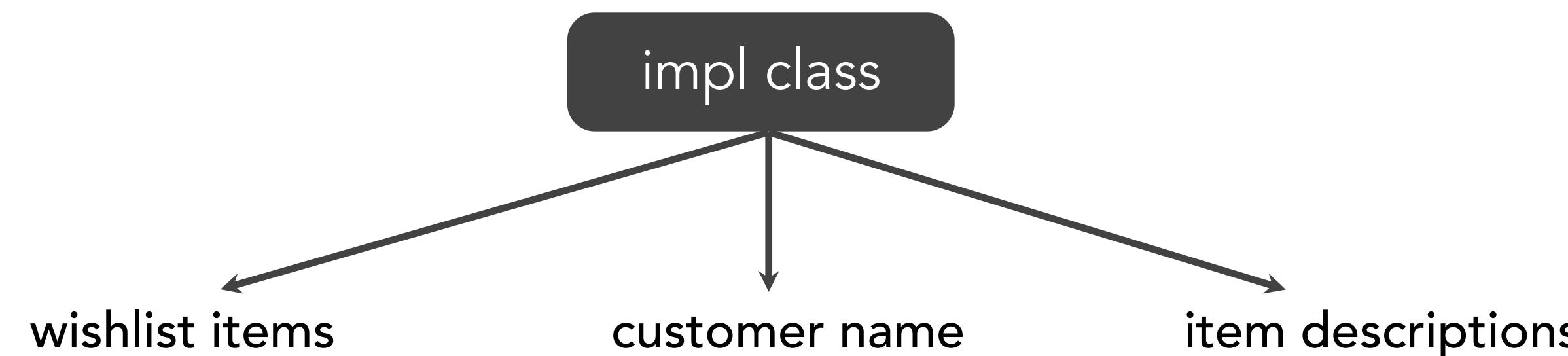
# service design

*micro-hexagonal design to support protocol agnostic processing*

## Implementation Class



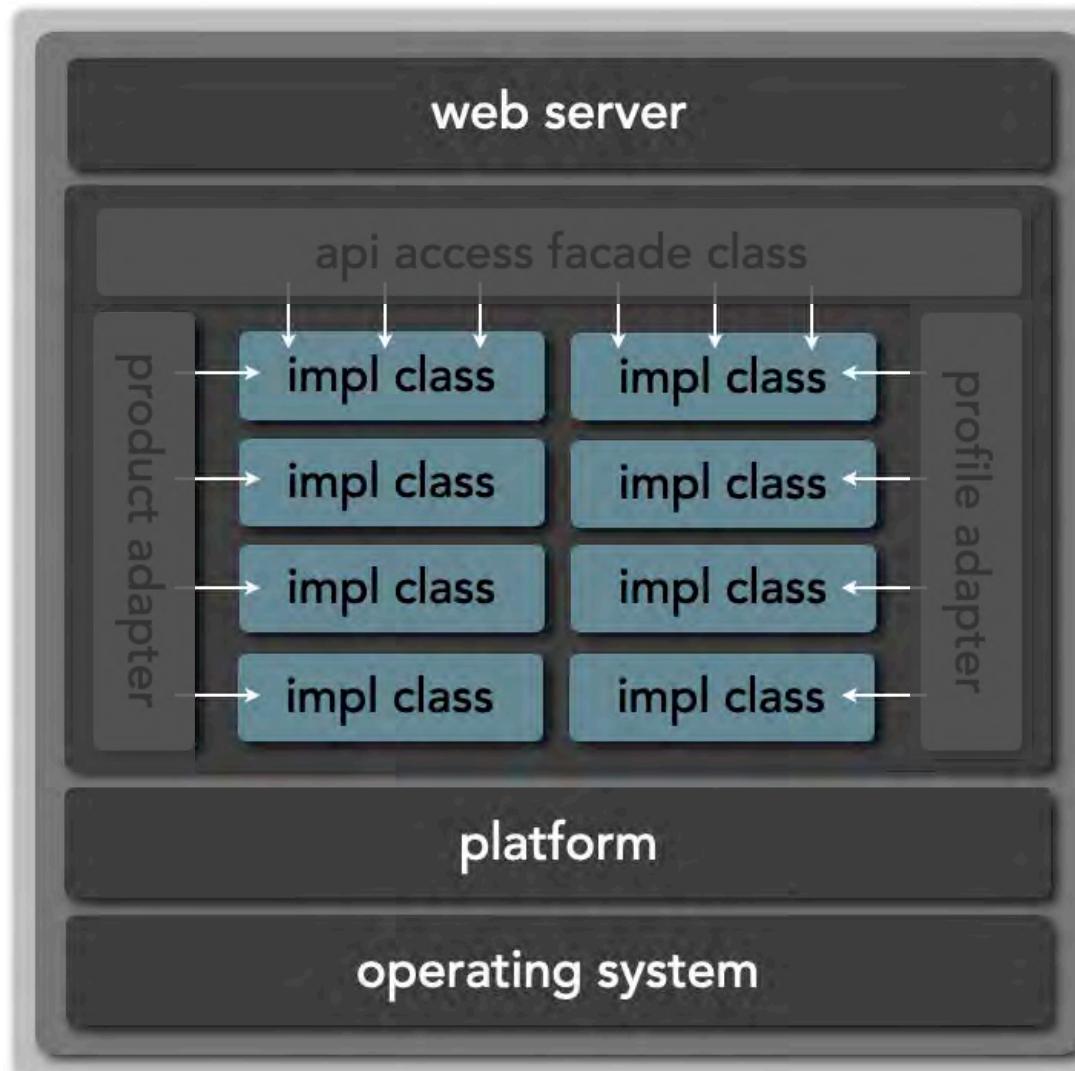
```
public WishlistItems getWishlistItems(long customerId) {  
    List<String> itemIds = getWishlistItemsFromDB(customerId);  
    String customerName = profileAdapter.getCustomerName(customerId);  
    Map<String, String> itemDescriptions =  
        productAdapter.getProductDescs(itemIds);  
    return new WishlistItems(customerName, itemDescriptions);  
}
```



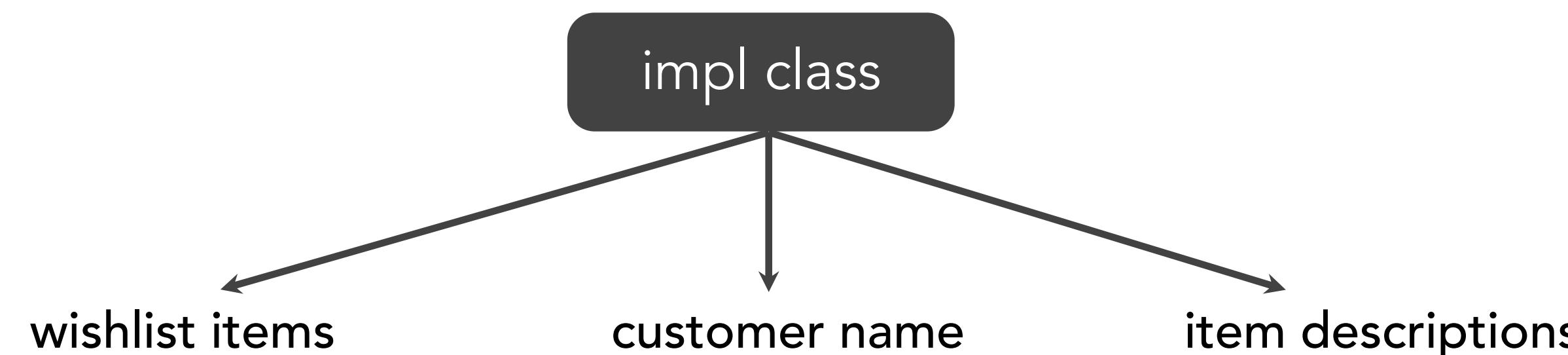
# service design

*micro-hexagonal design to support protocol agnostic processing*

## Implementation Class



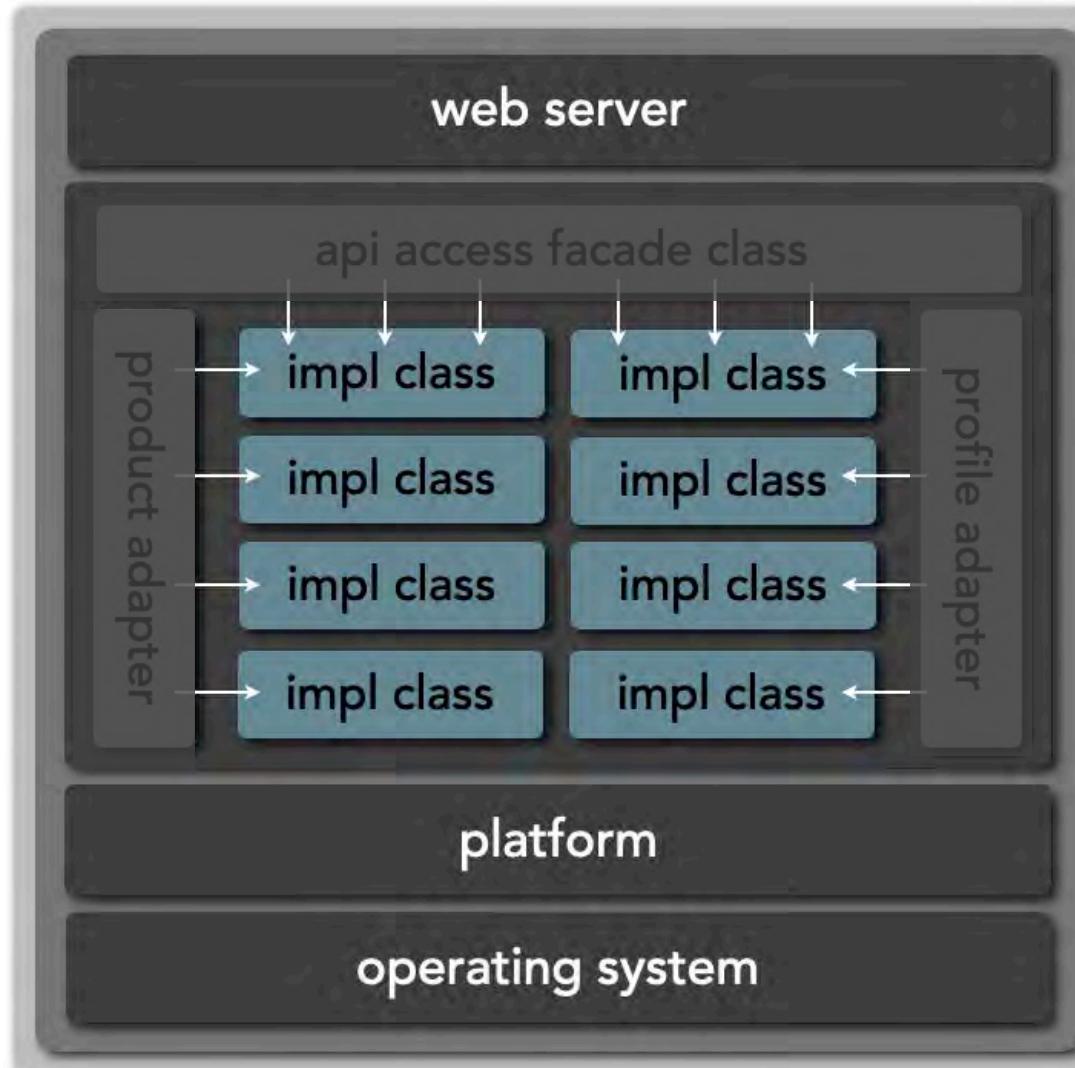
```
public WishlistItems getWishlistItems(long customerId) {  
    List<String> itemIds = getWishlistItemsFromDB(customerId);  
    String customerName = profileAdapter.getCustomerName(customerId);  
    Map<String, String> itemDescriptions =  
        productAdapter.getProductDescs(itemIds);  
    return new WishlistItems(customerName, itemDescriptions);  
}
```



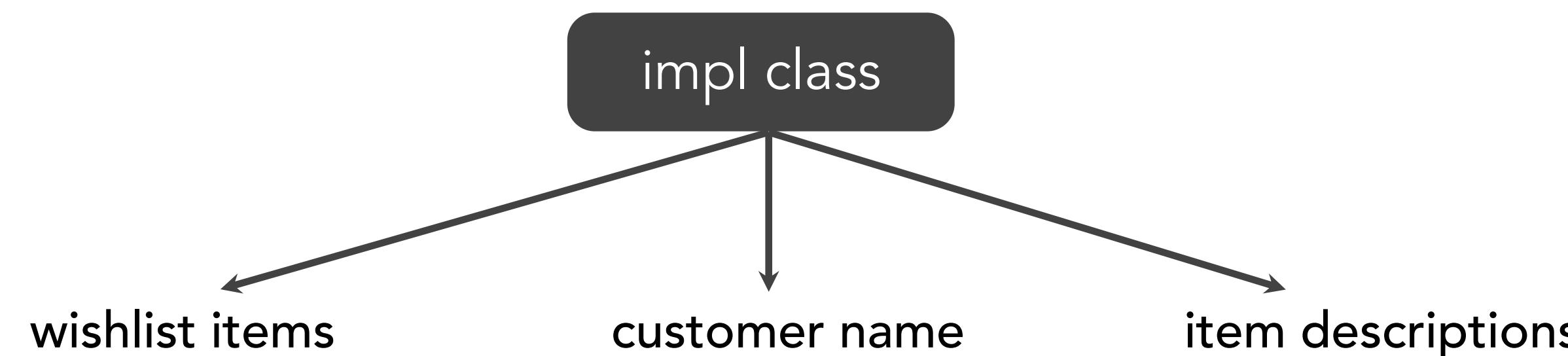
# service design

*micro-hexagonal design to support protocol agnostic processing*

## Implementation Class



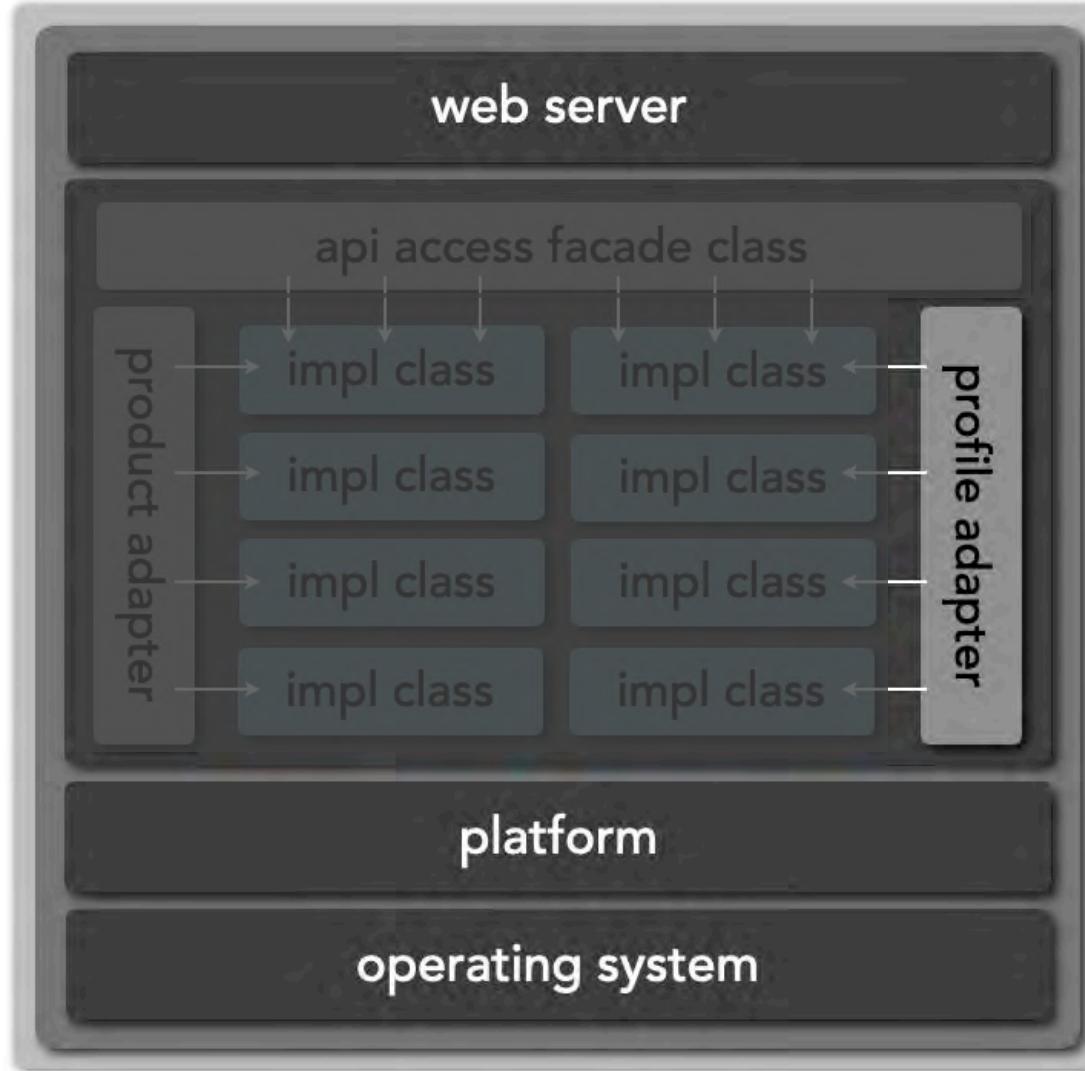
```
public WishlistItems getWishlistItems(long customerId) {  
    List<String> itemIds = getWishlistItemsFromDB(customerId);  
    String customerName = profileAdapter.getCustomerName(customerId);  
    Map<String, String> itemDescriptions =  
        productAdapter.getProductDescs(itemIds);  
    return new WishlistItems(customerName, itemDescriptions);  
}
```



# service design

*micro-hexagonal design to support protocol agnostic processing*

## Profile Adapter Class (AMQP)



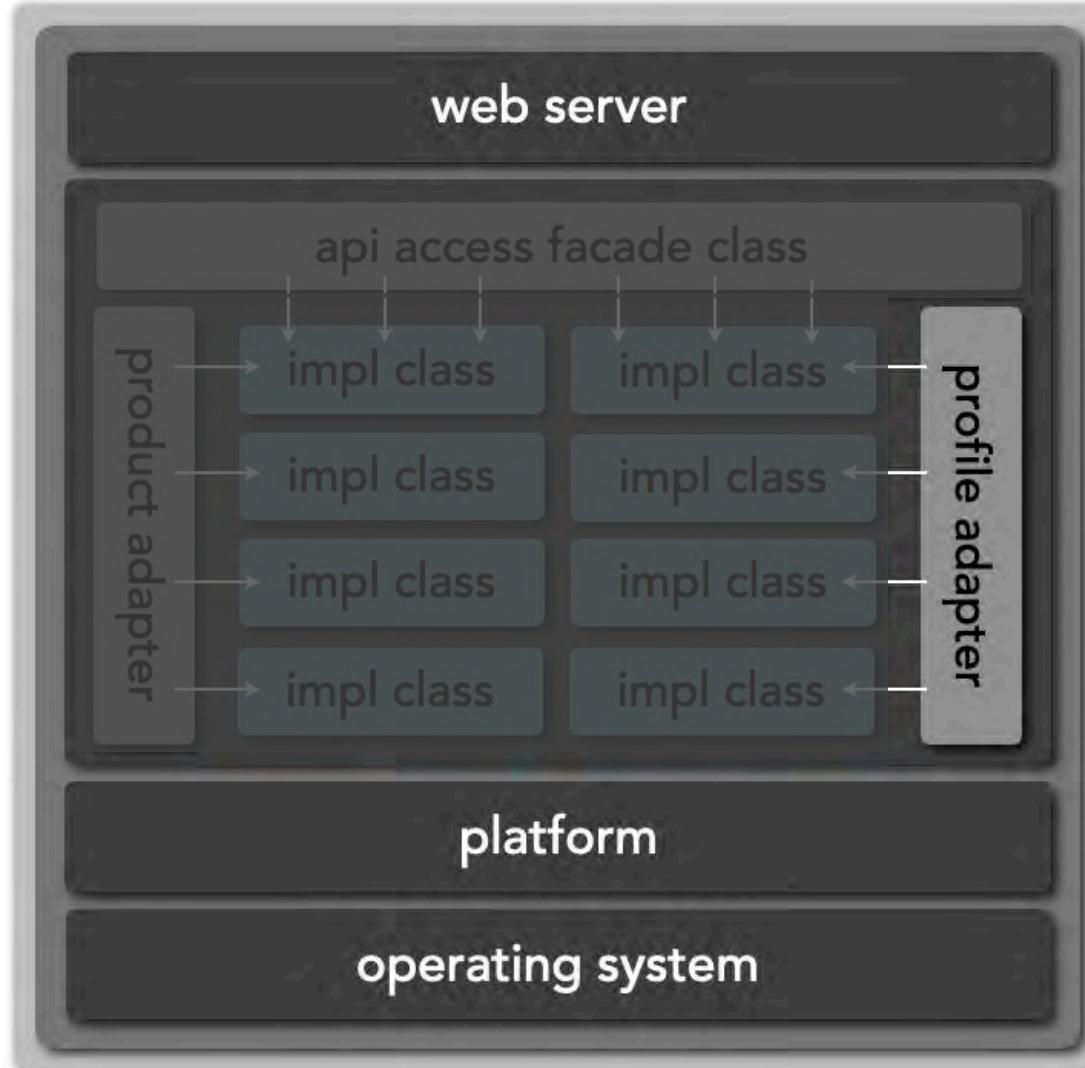
```
public String getCustomerName(long customerId) {
    Channel channel = connection.createChannel();
    byte[] message = new Long(customerId).toString().getBytes();
    BasicProperties props = new BasicProperties
        .Builder().replyTo("profile.getname.response.q").build();
    channel.basicPublish("", "profile.getname.request.q", props, message);

    final BlockingQueue<String> response = new ArrayBlockingQueue<String>(1);
    channel.basicConsume("profile.getname.response.q", true,
        new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                response.offer(new String(body, "UTF-8"));
            }
        });
    String name = new String(response.take());
    channel.close();
    return name;
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## Profile Adapter Class (AMQP)



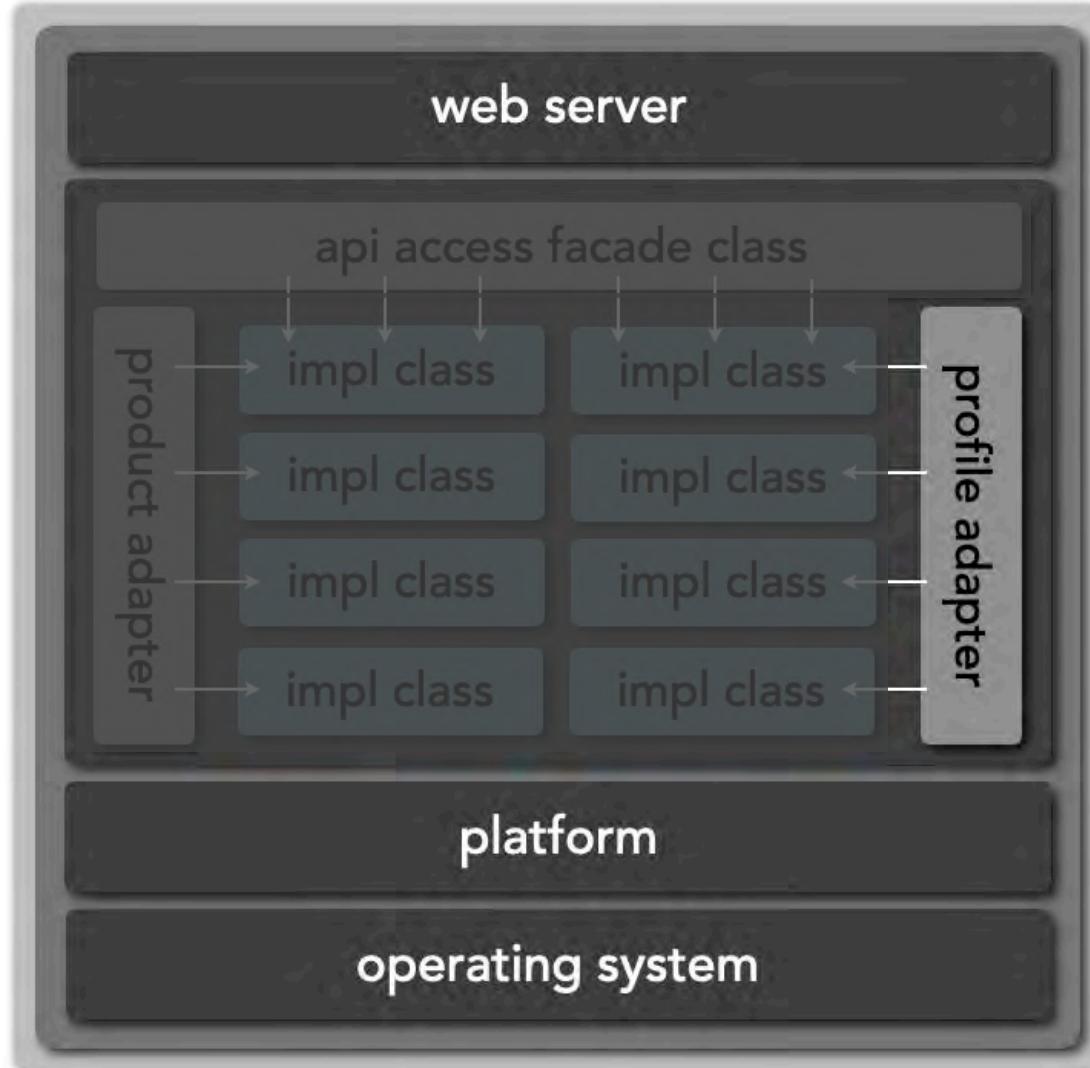
```
public String getCustomerName(long customerId) {
    Channel channel = connection.createChannel();
    byte[] message = new Long(customerId).toString().getBytes();
    BasicProperties props = new BasicProperties
        .Builder().replyTo("profile.getname.response.q").build();
    channel.basicPublish("", "profile.getname.request.q", props, message);

    final BlockingQueue<String> response = new ArrayBlockingQueue<String>(1);
    channel.basicConsume("profile.getname.response.q", true,
        new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                response.offer(new String(body, "UTF-8"));
            }
        });
    String name = new String(response.take());
    channel.close();
    return name;
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## Profile Adapter Class (AMQP)



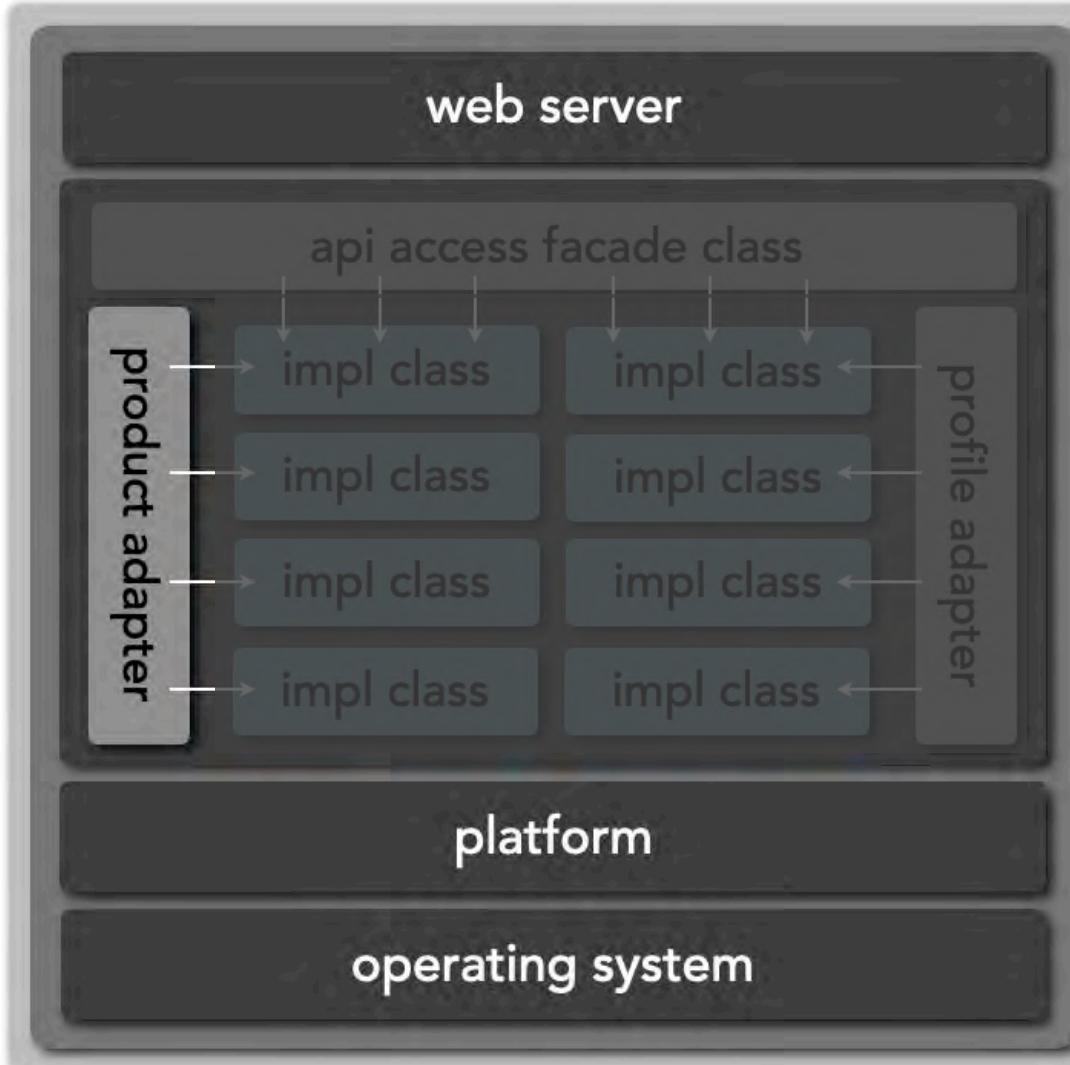
```
public String getCustomerName(long customerId) {
    Channel channel = connection.createChannel();
    byte[] message = new Long(customerId).toString().getBytes();
    BasicProperties props = new BasicProperties
        .Builder().replyTo("profile.getname.response.q").build();
    channel.basicPublish("", "profile.getname.request.q", props, message);

    final BlockingQueue<String> response = new ArrayBlockingQueue<String>(1);
    channel.basicConsume("profile.getname.response.q", true,
        new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                response.offer(new String(body, "UTF-8"));
            }
        });
    String name = new String(response.take());
    channel.close();
    return name;
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## Product Adapter Class (gRPC)

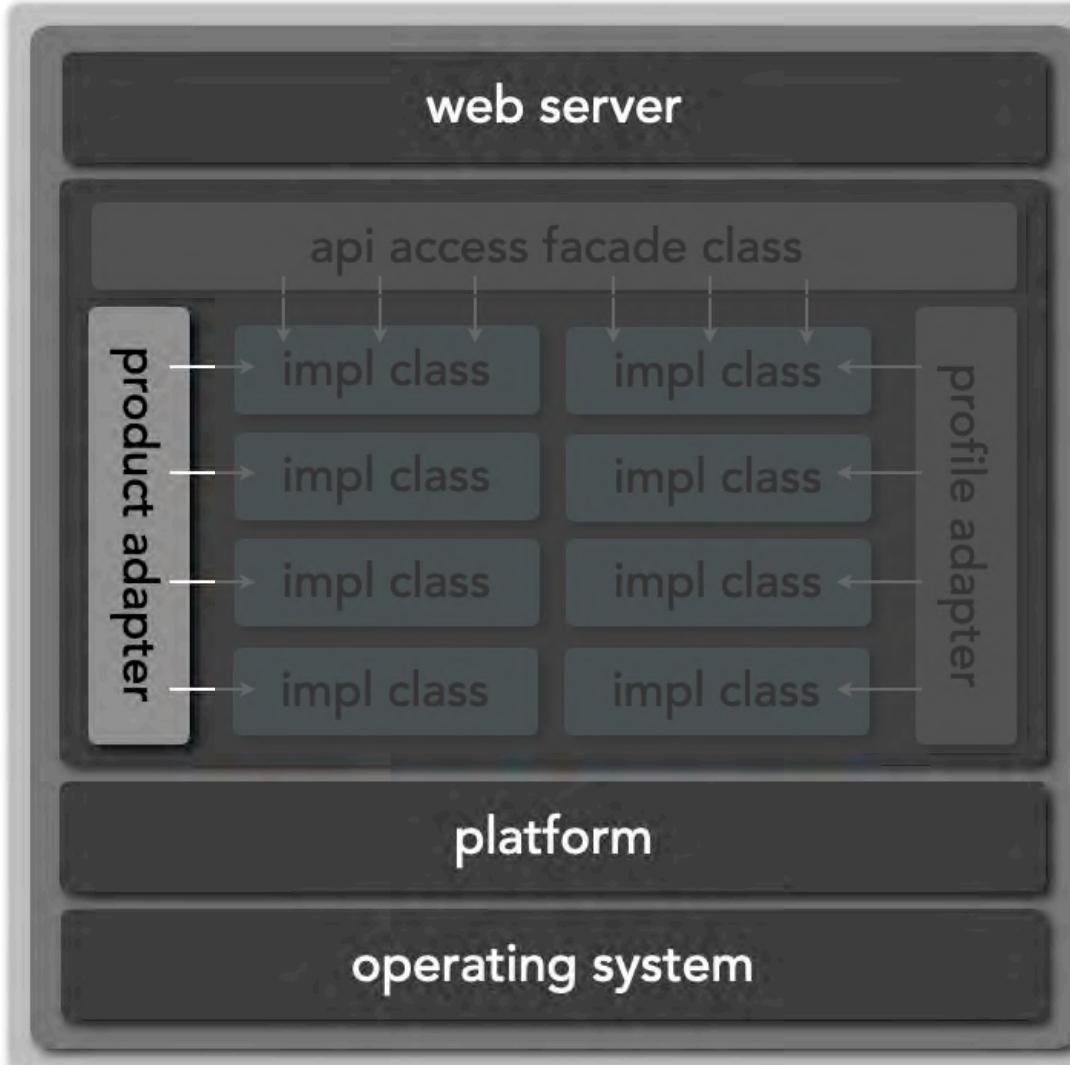


```
public List<String> getProductDescs(List<String> productIds) {
    ProductRequest request = ProductRequest.newBuilder()
        .setProductIds(productIds).build();
    ProductReply response;
    try {
        response = blockingStub.getDescriptions(request);
    } catch (StatusRuntimeException e) {
        logger.log(Level.ERROR, "RPC call failed: {0}", e.getStatus());
        return new ArrayList<String>();
    }
    return response.getMessage();
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

## Product Adapter Class (gRPC)

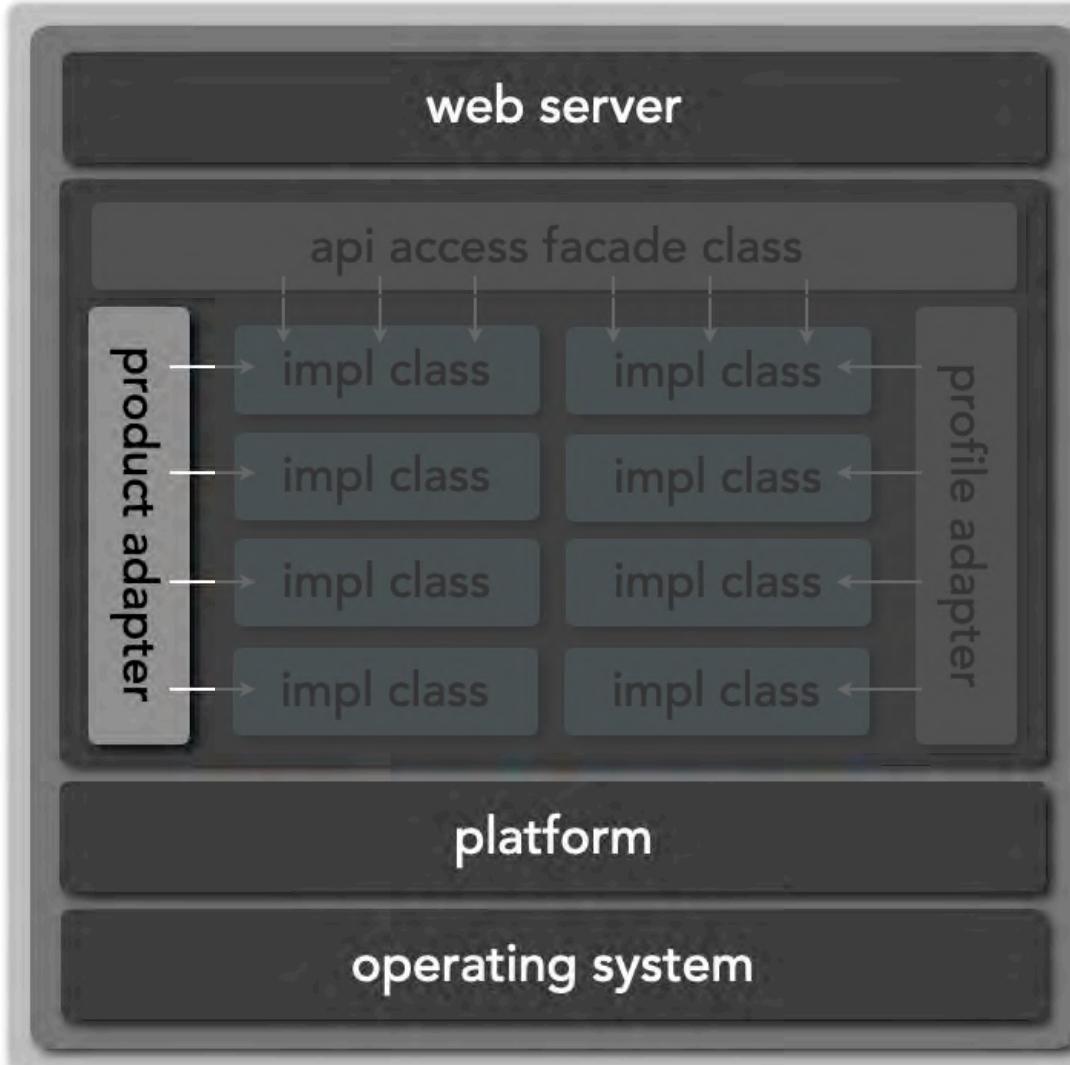


```
public List<String> getProductDescs(List<String> productIds) {  
    ProductRequest request = ProductRequest.newBuilder()  
        .setProductIds(productIds).build();  
    ProductReply response;  
    try {  
        response = blockingStub.getDescriptions(request);  
    } catch (StatusRuntimeException e) {  
        logger.log(Level.ERROR, "RPC call failed: {0}", e.getStatus());  
        return new ArrayList<String>();  
    }  
    return response.getMessage();  
}
```

# service design

*micro-hexagonal design to support protocol agnostic processing*

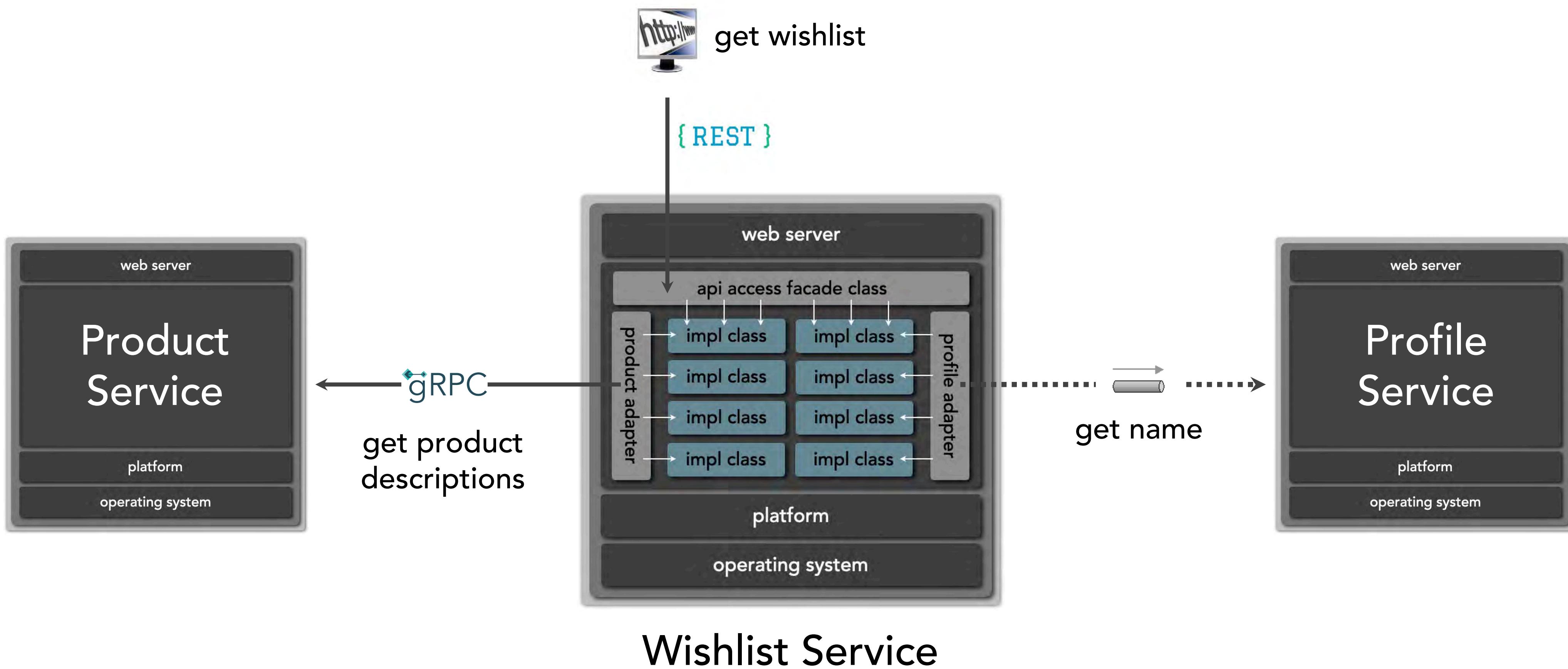
## Product Adapter Class (gRPC)



```
public List<String> getProductDescs(List<String> productIds) {
    ProductRequest request = ProductRequest.newBuilder()
        .setProductIds(productIds).build();
    ProductReply response;
    try {
        response = blockingStub.getDescriptions(request);
    } catch (StatusRuntimeException e) {
        logger.log(Level.ERROR, "RPC call failed: {0}", e.getStatus());
        return new ArrayList<String>();
    }
    return response.getMessage();
}
```

# service design

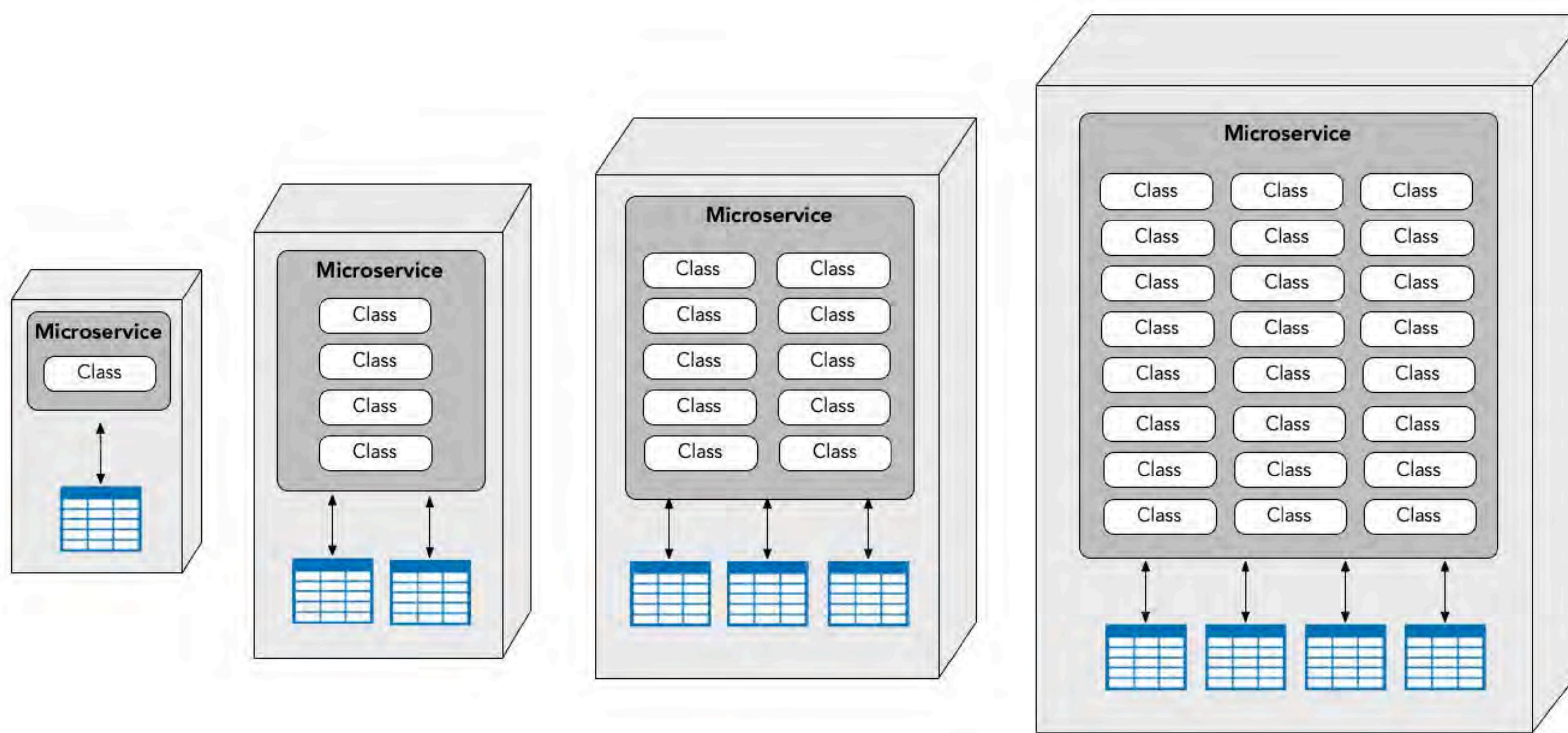
*micro-hexagonal design to support protocol agnostic processing*



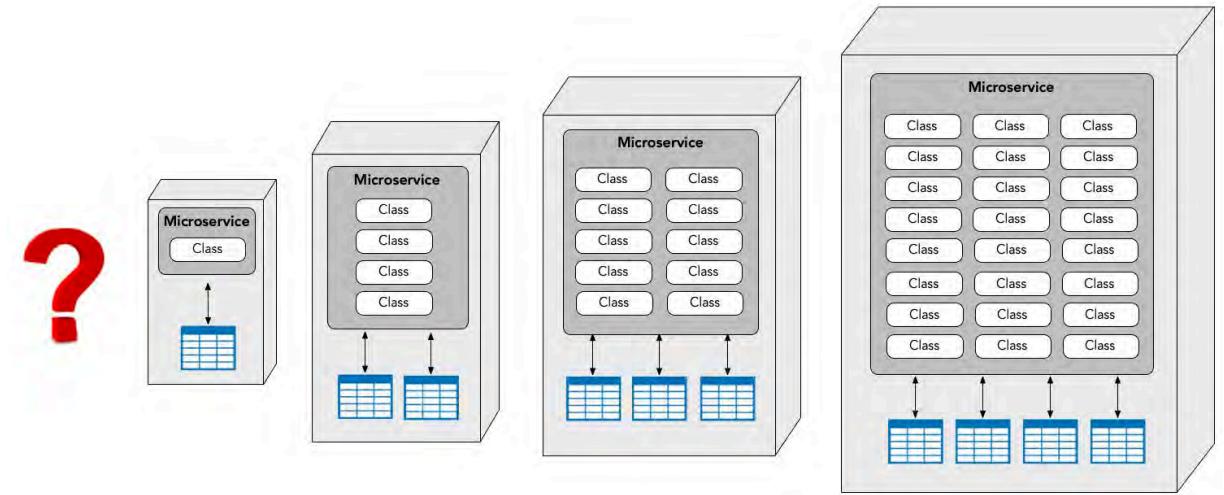
# Service Granularity

# what is the right level of granularity for a service?

?



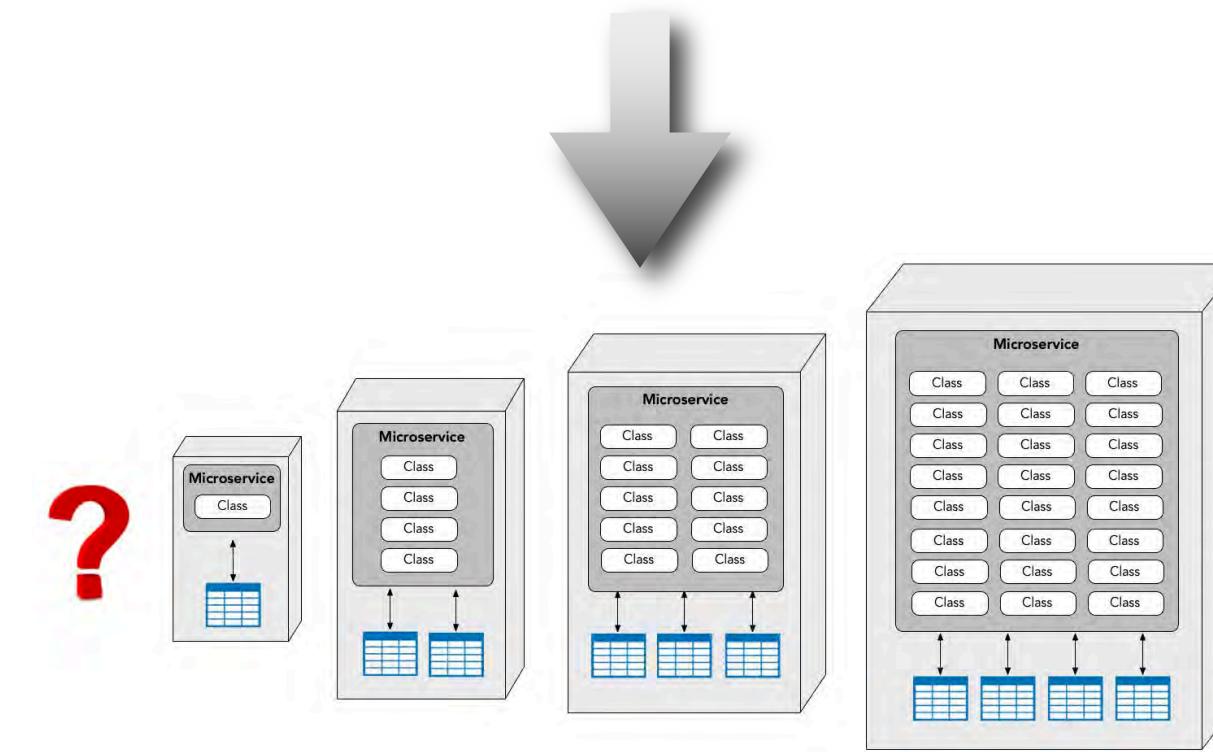
# service granularity



# service granularity

## granularity disintegrators

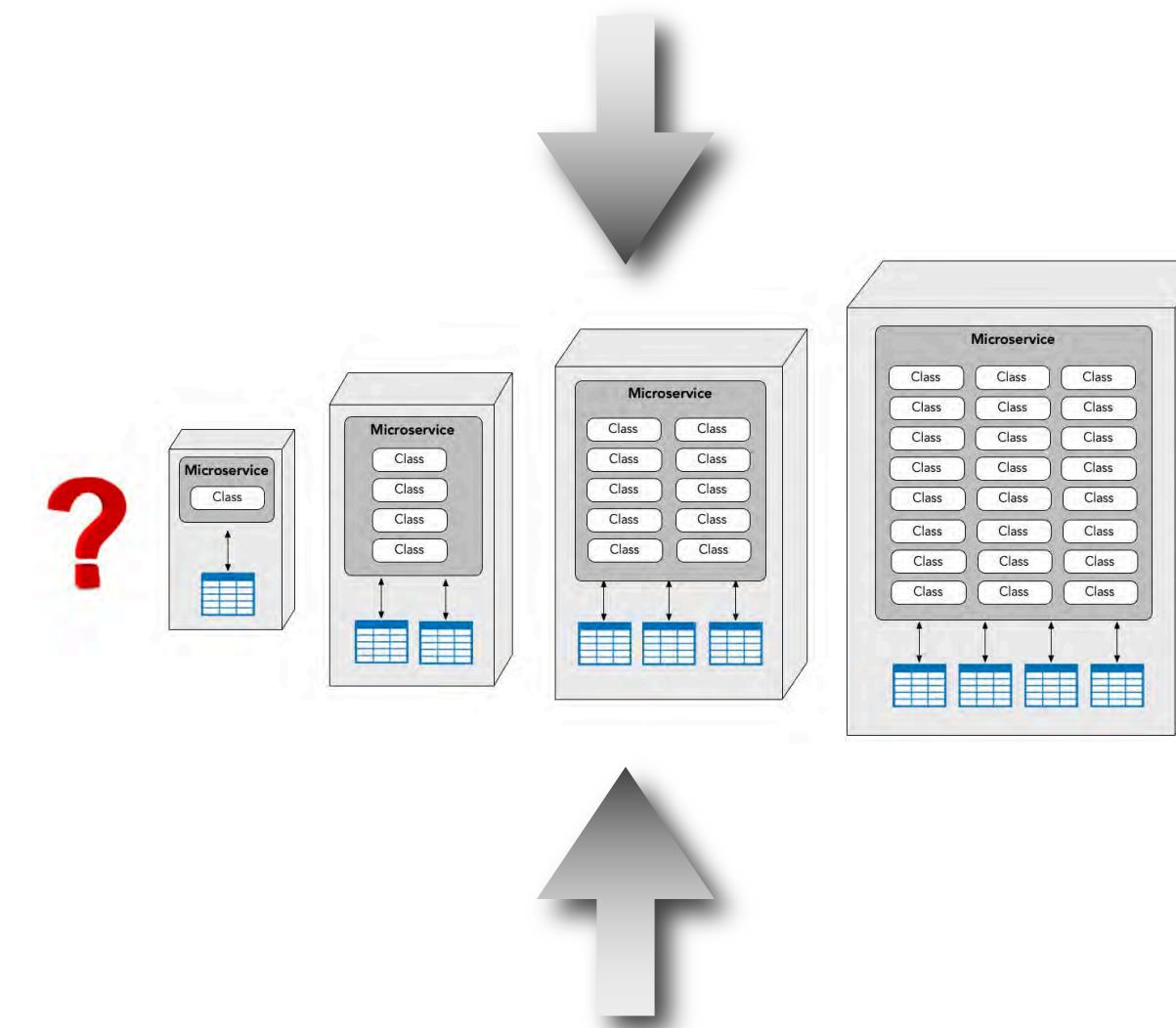
*“when should I consider breaking apart a service?”*



# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



## granularity integrators

*“when should I consider putting services back together?”*

# service granularity

granularity disintegrators

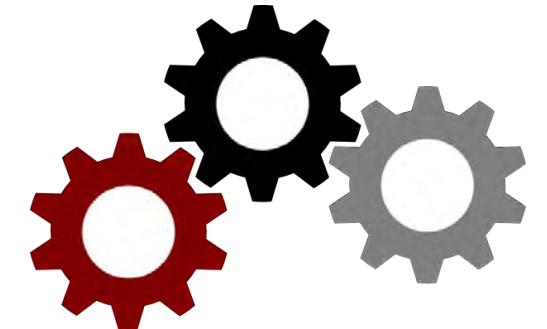
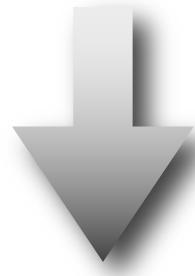
*“when should I consider breaking apart a service?”*



# service granularity

**granularity disintegrators**

*“when should I consider breaking apart a service?”*

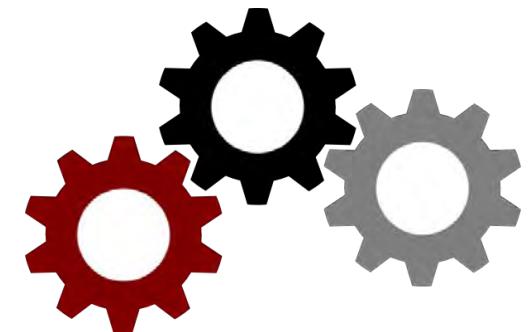


service  
functionality

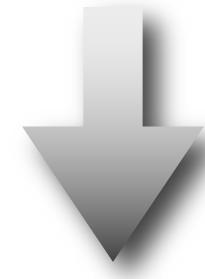
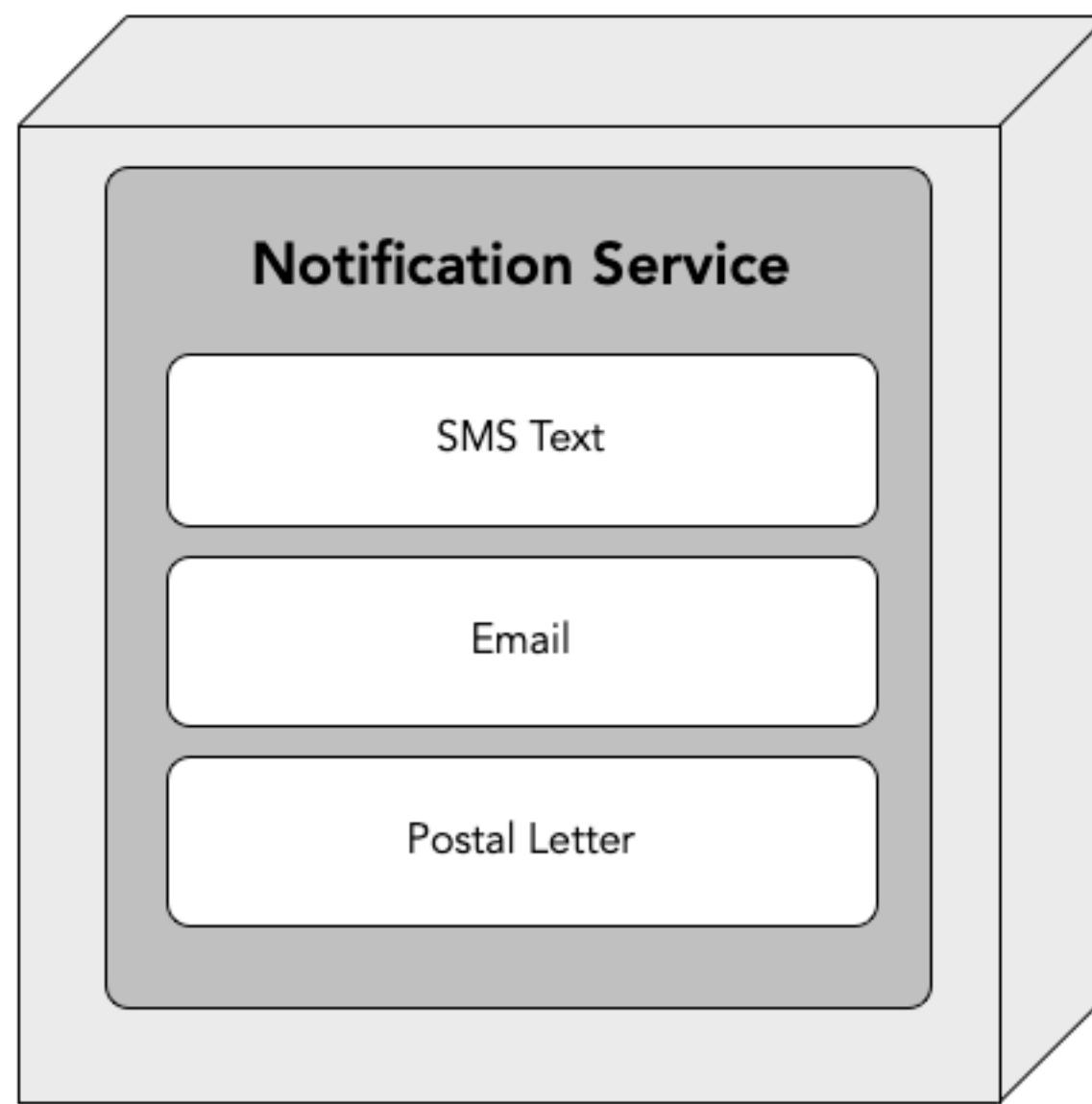
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



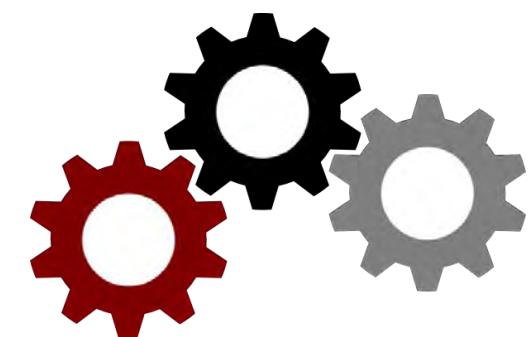
service  
functionality



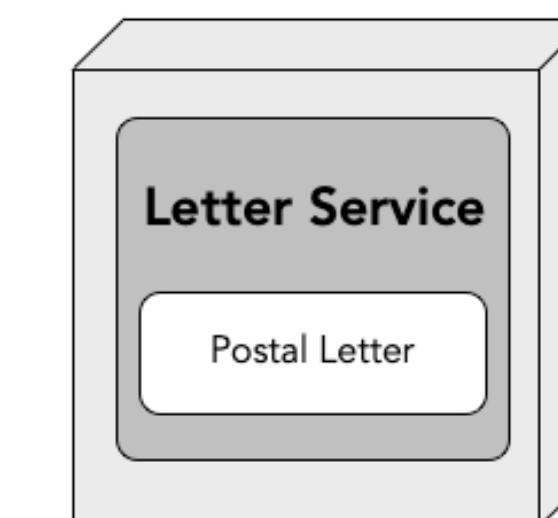
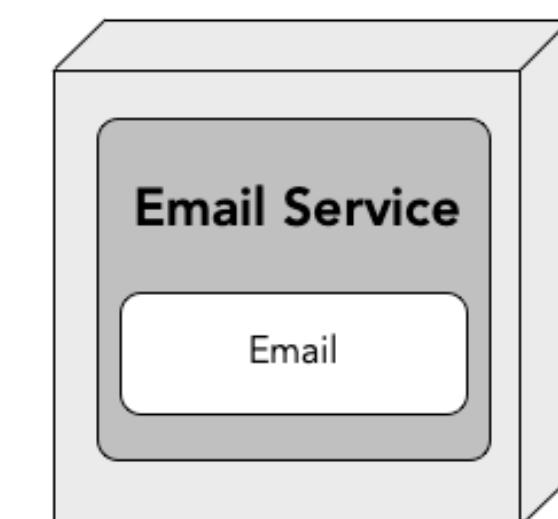
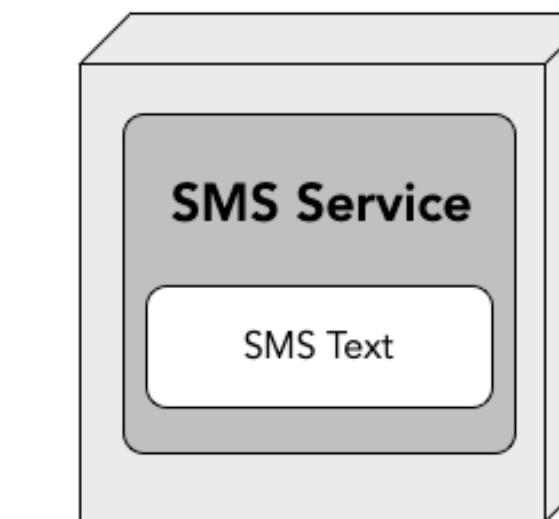
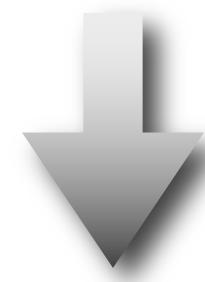
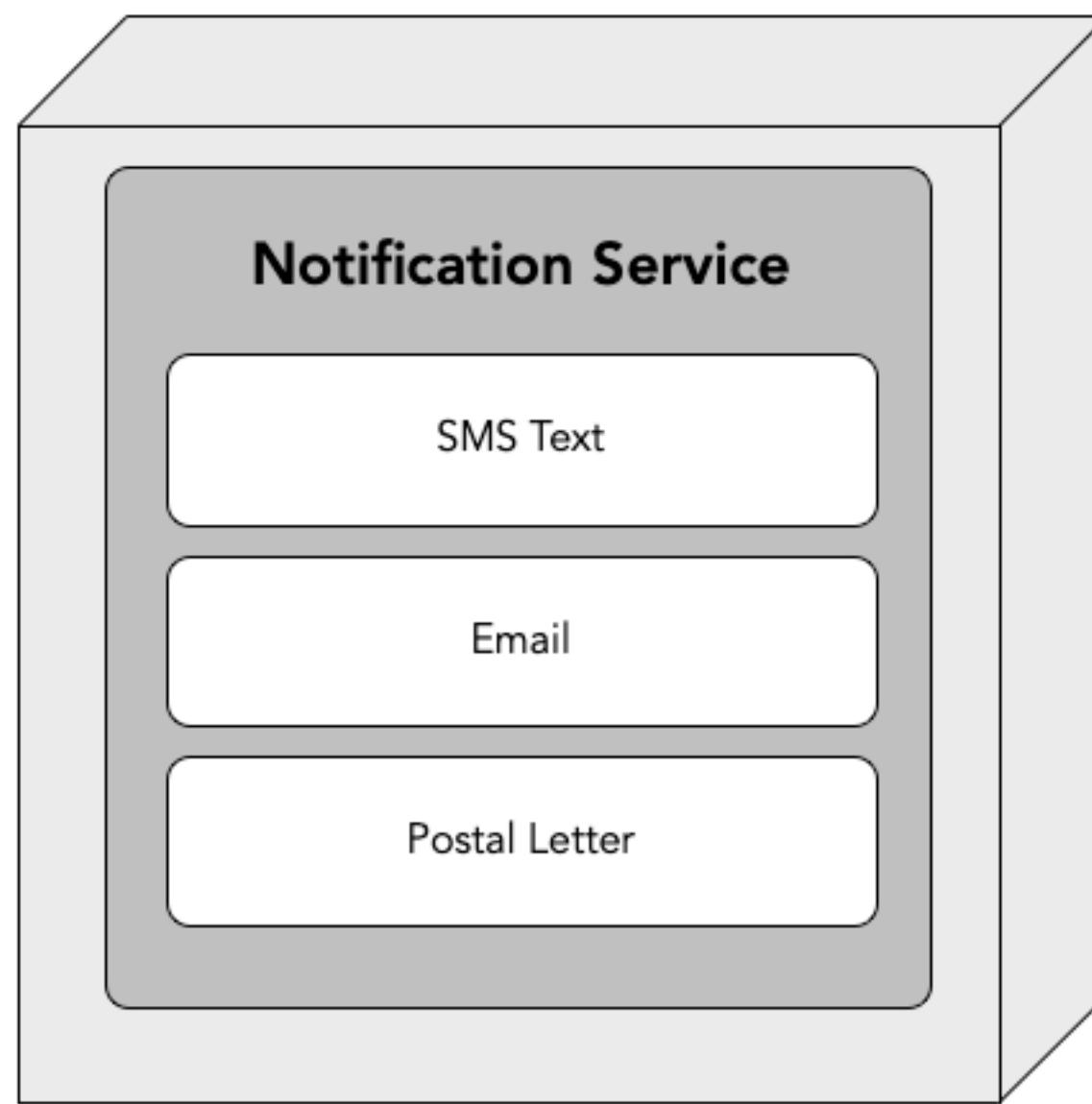
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



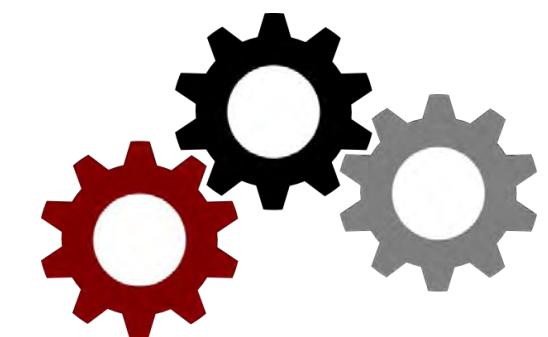
service  
functionality



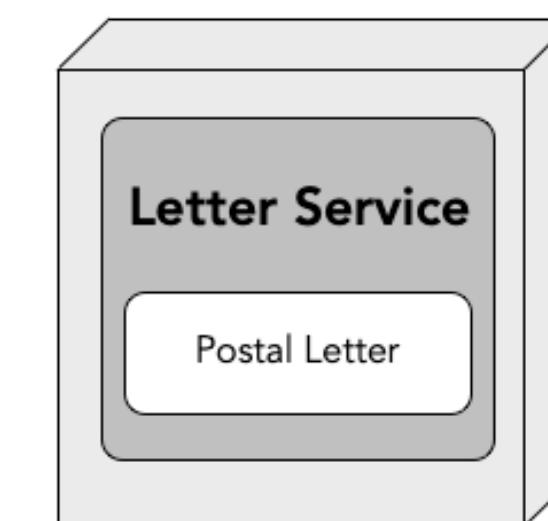
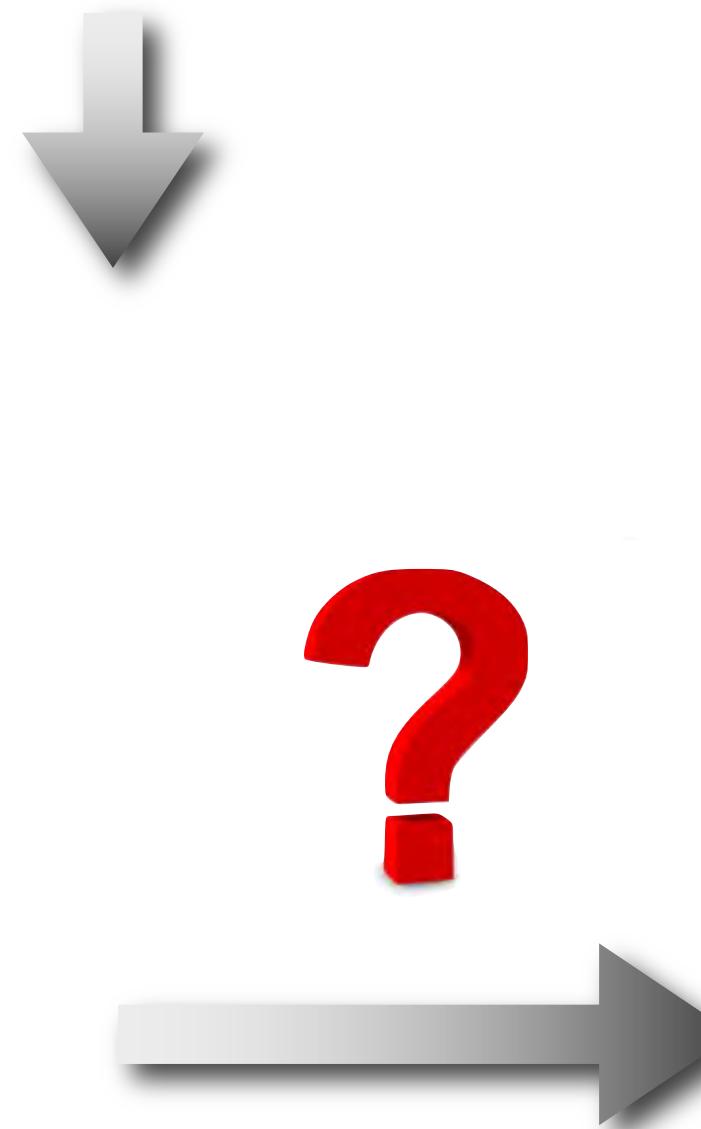
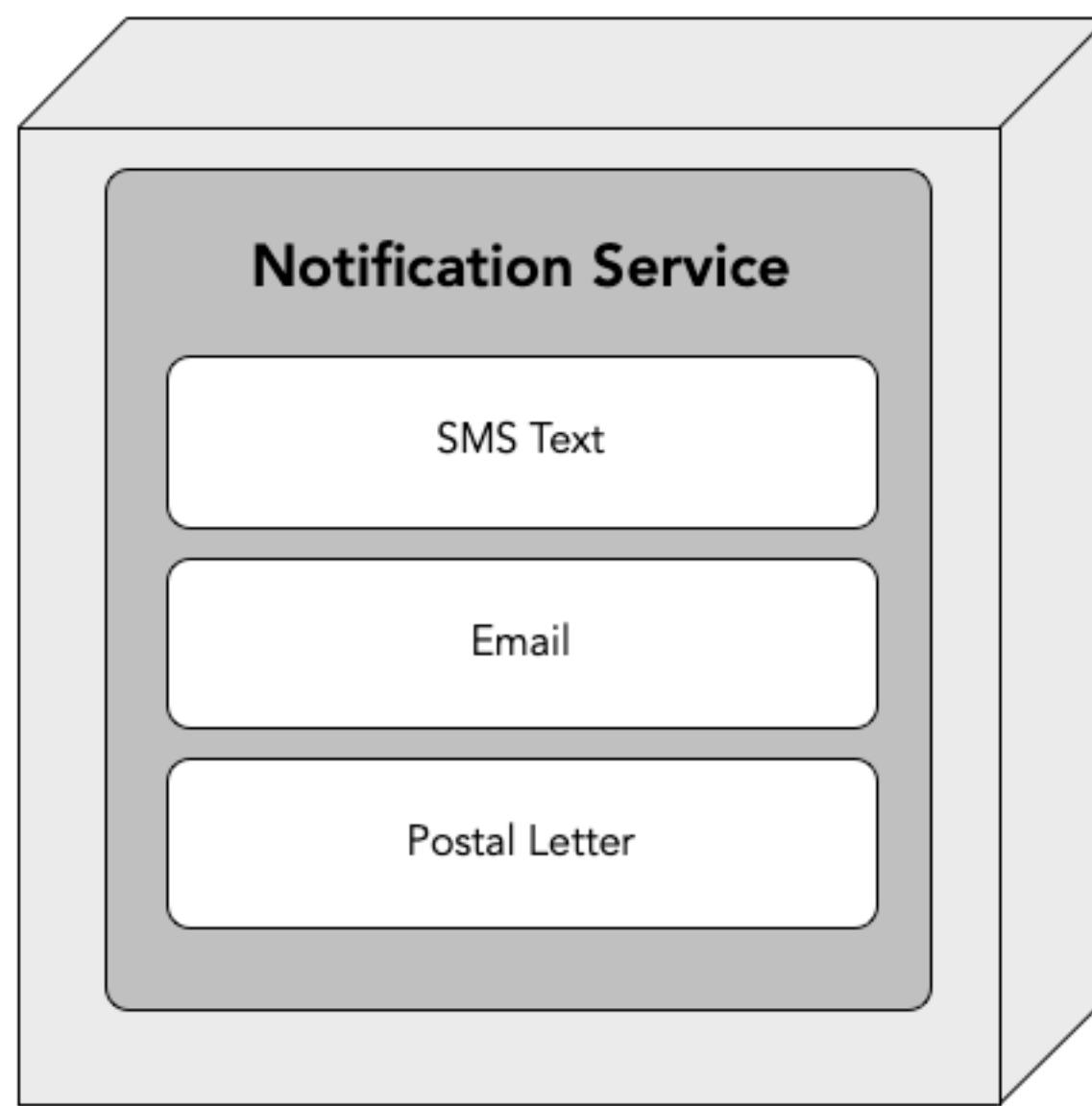
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



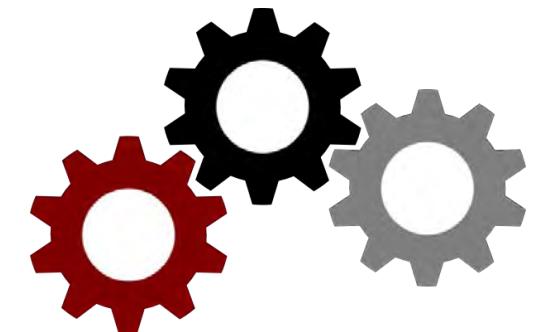
service  
functionality



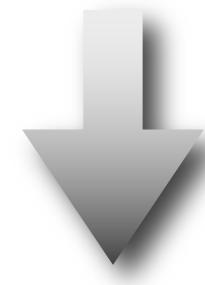
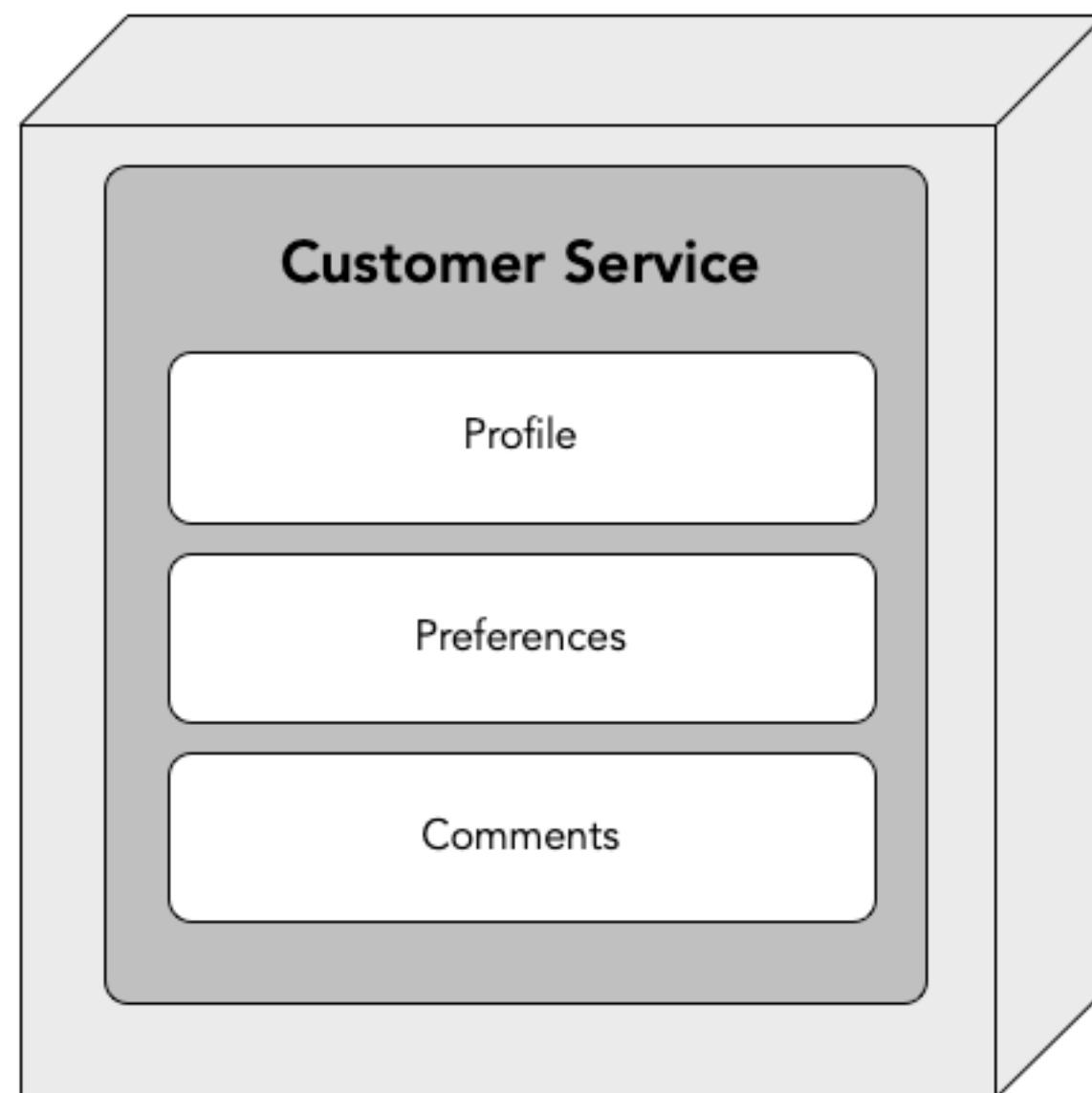
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



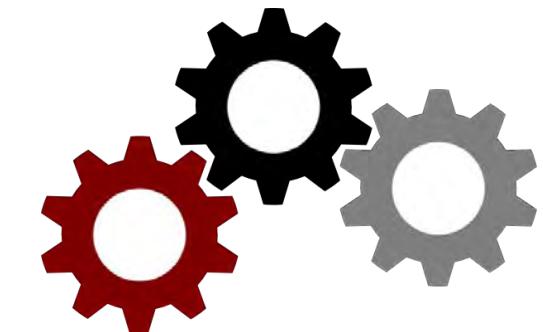
service  
functionality



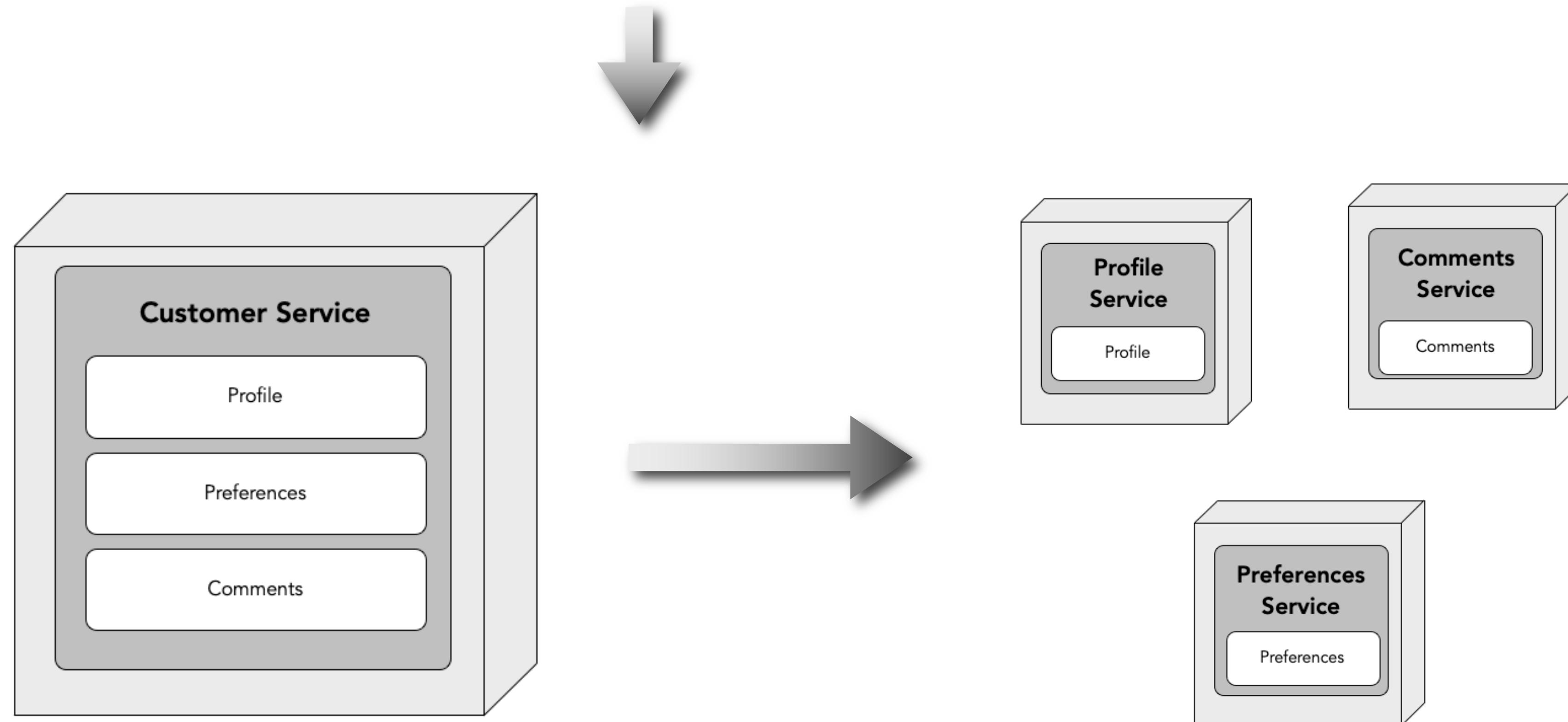
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



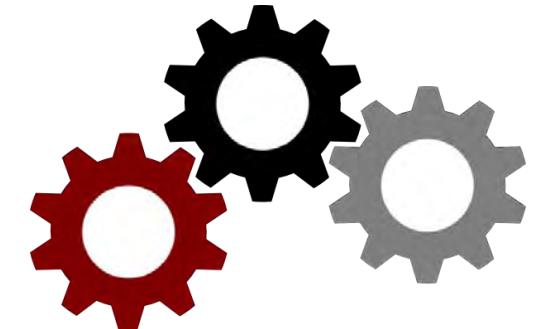
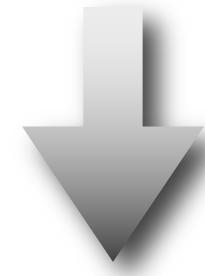
service  
functionality



# service granularity

**granularity disintegrators**

*“when should I consider breaking apart a service?”*

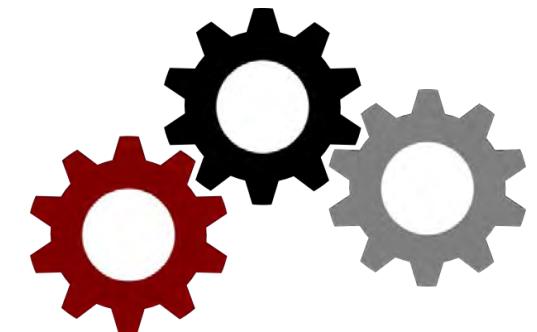
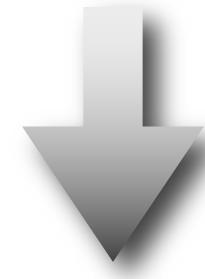


service  
functionality

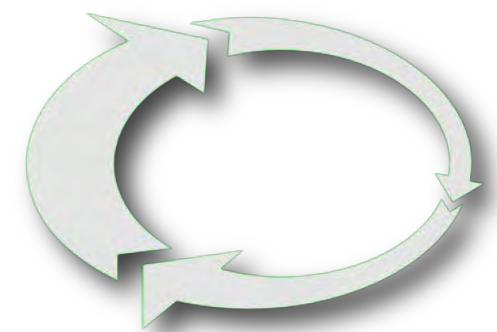
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



service  
functionality

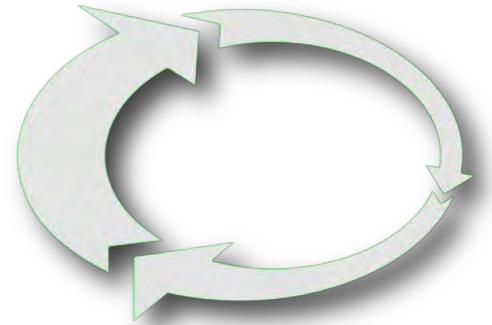


code  
volatility

# service granularity

granularity disintegrators

*“when should I consider breaking apart a service?”*

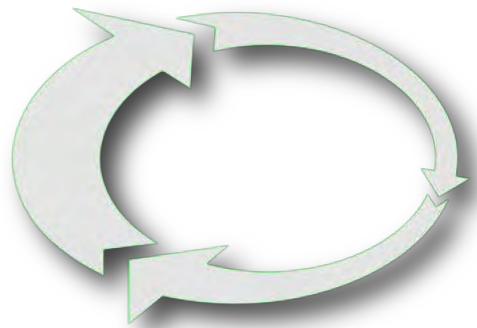


code  
volatility

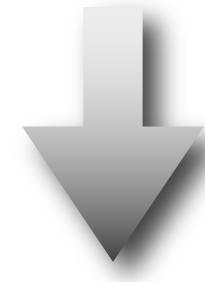
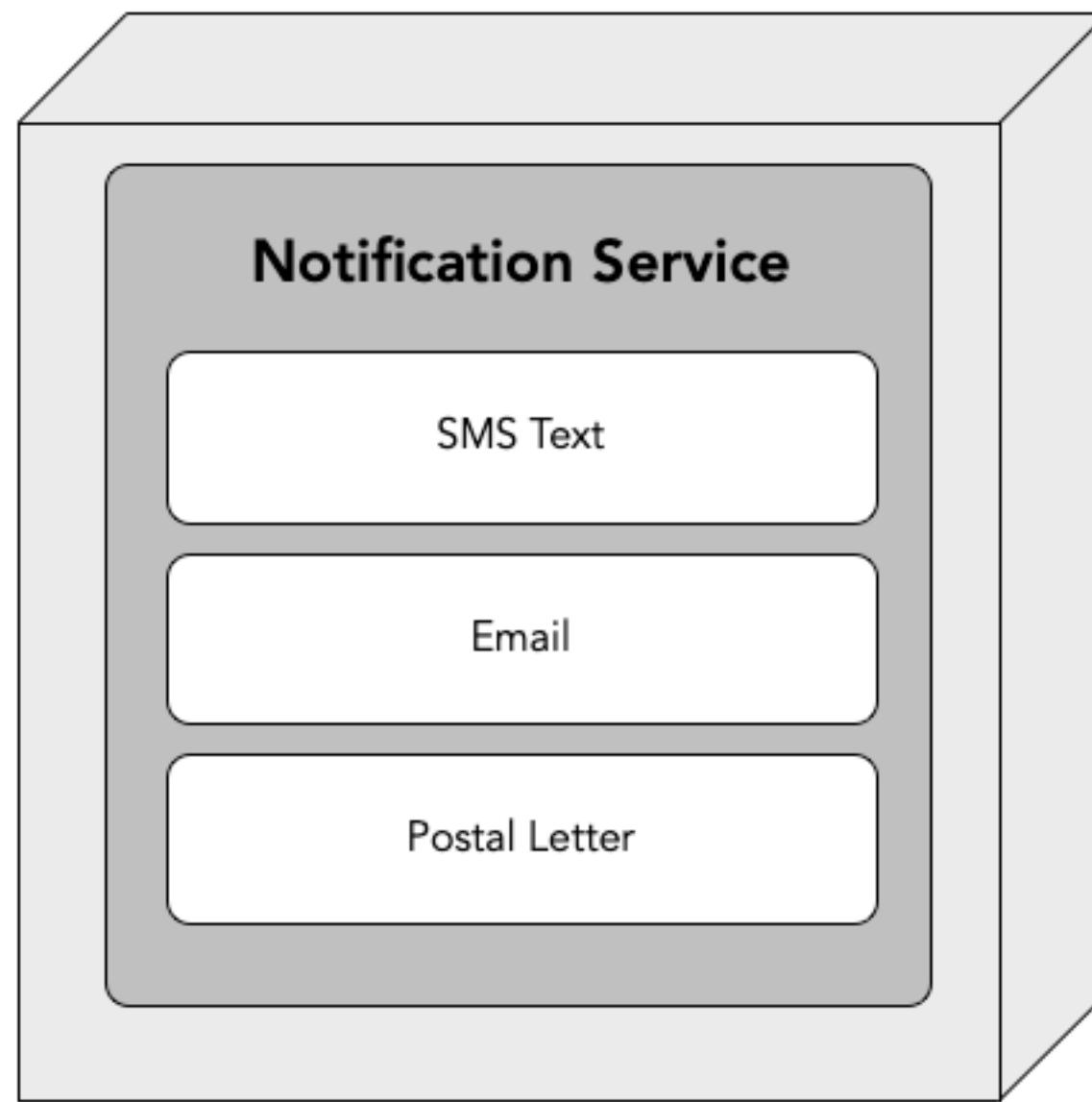
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



code  
volatility

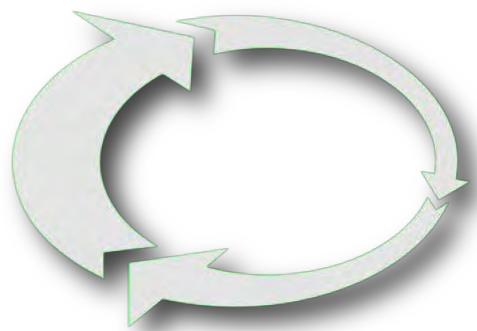


*“when should I consider breaking apart a service?”*

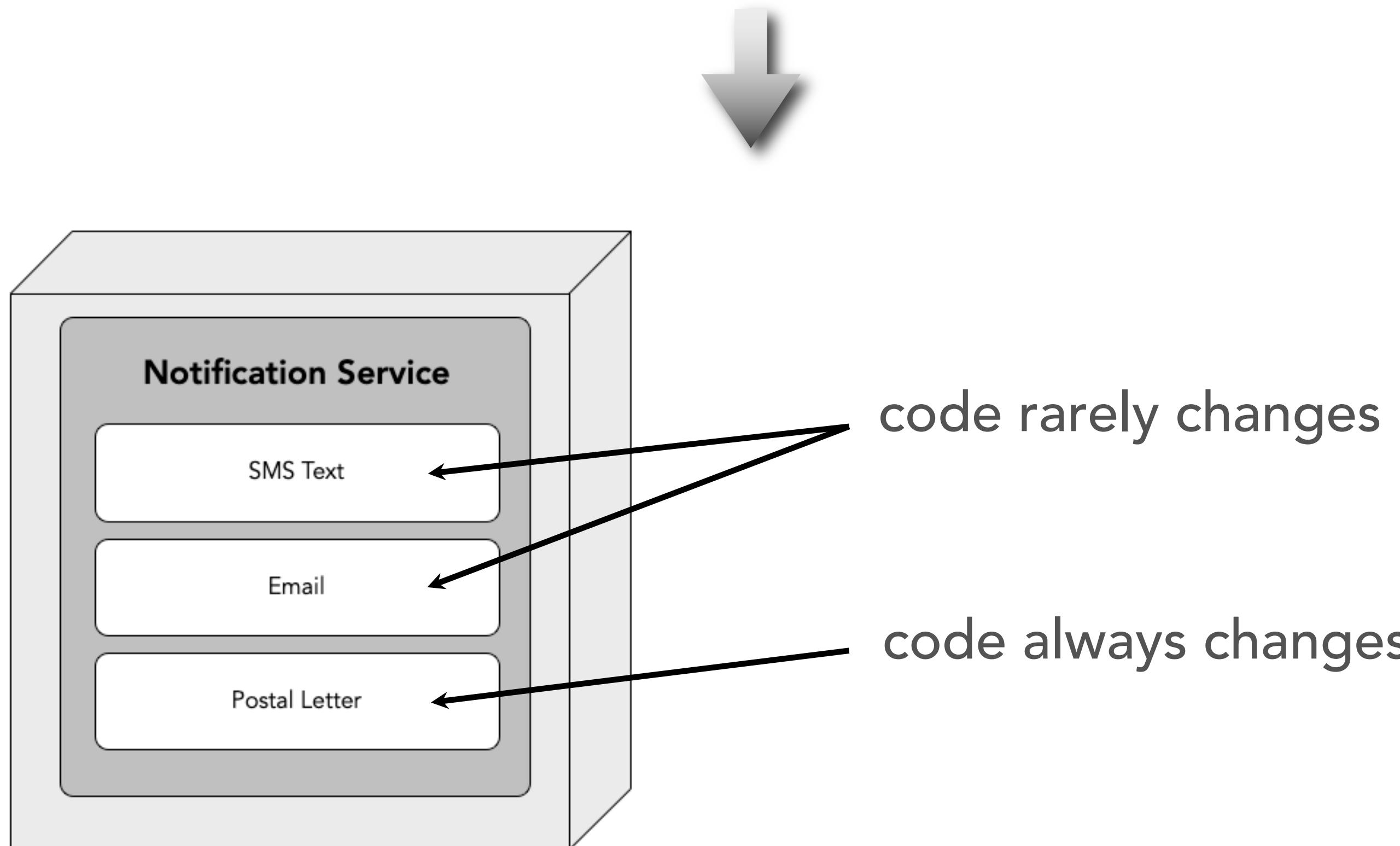
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



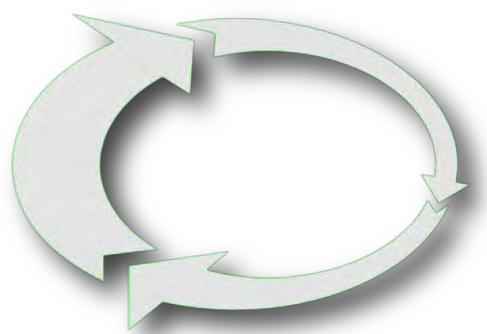
code  
volatility



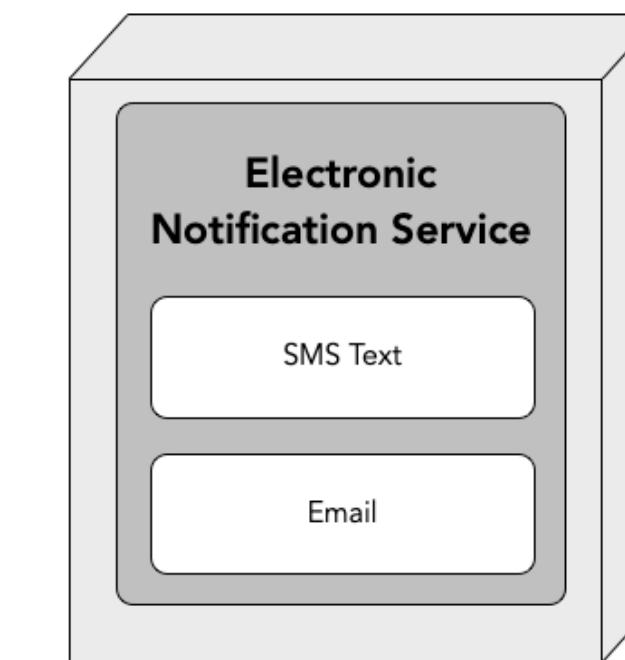
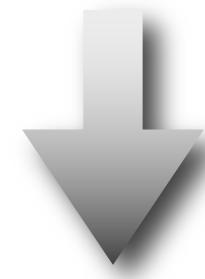
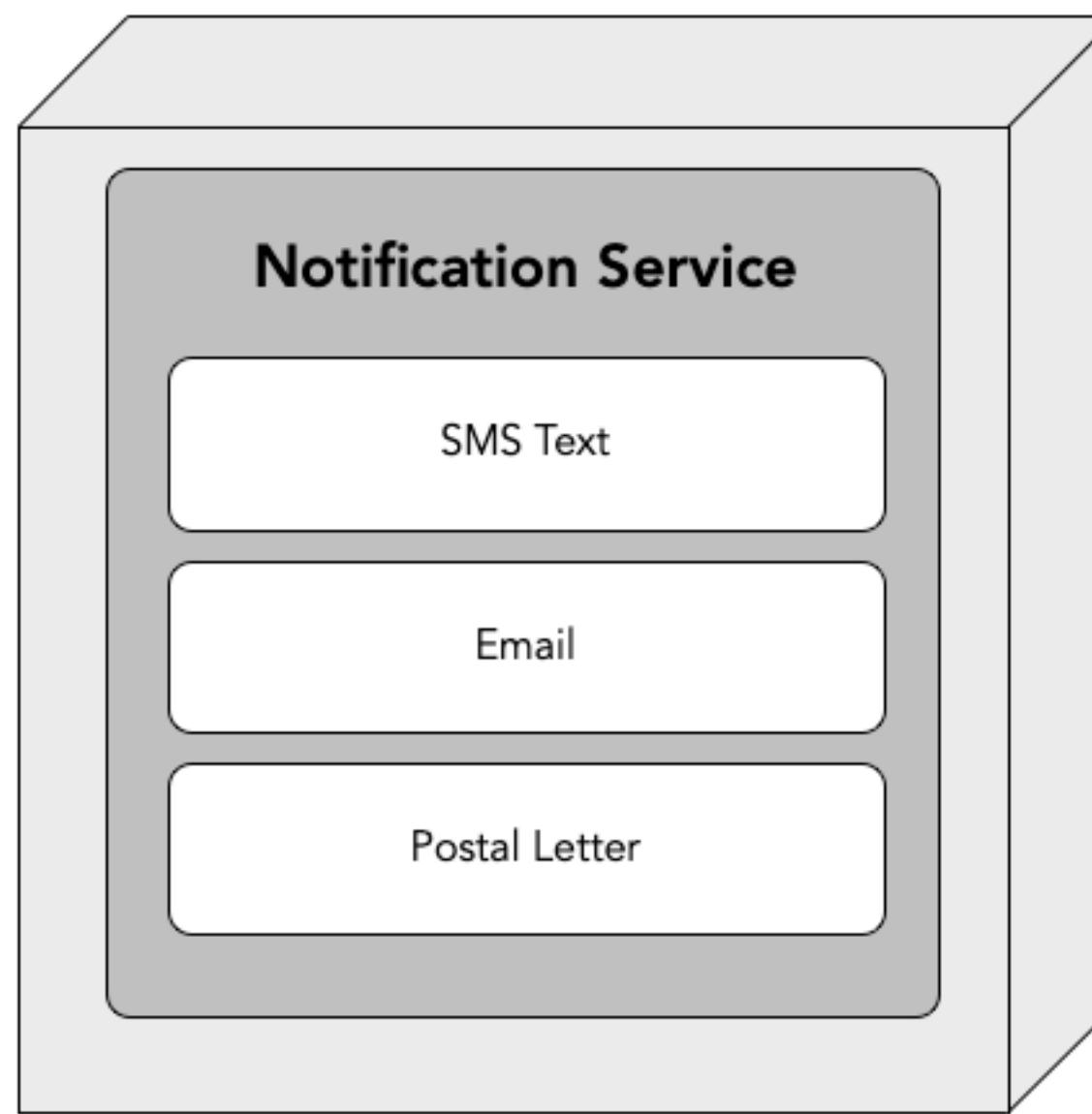
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



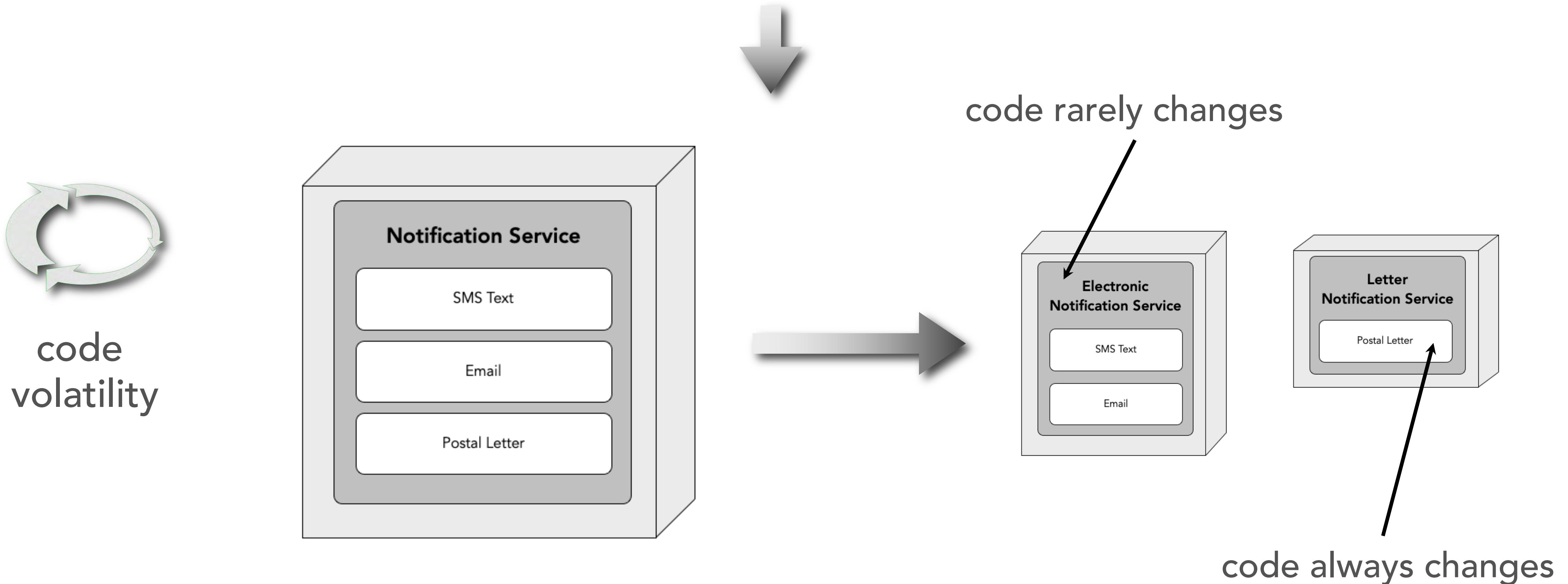
code  
volatility



# service granularity

## granularity disintegrators

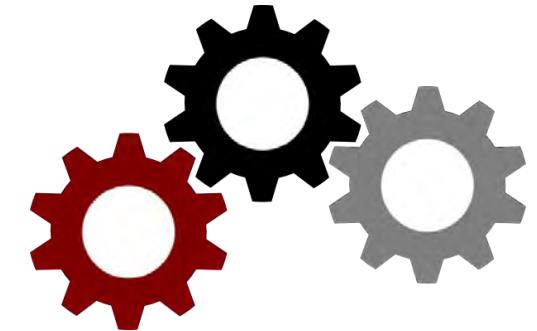
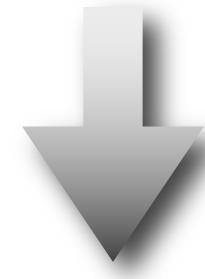
*“when should I consider breaking apart a service?”*



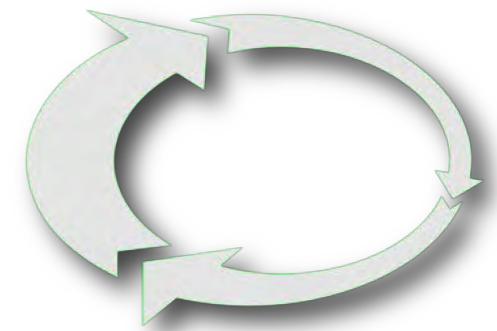
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



service  
functionality

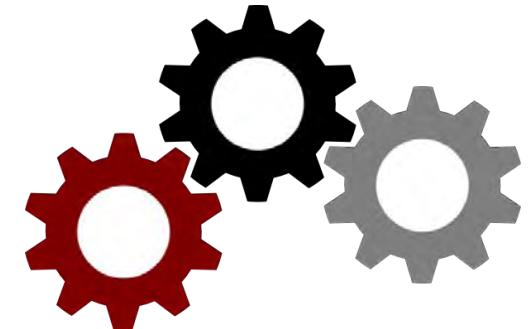


code  
volatility

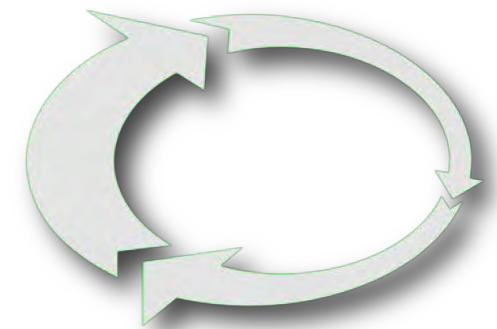
# service granularity

## granularity disintegrators

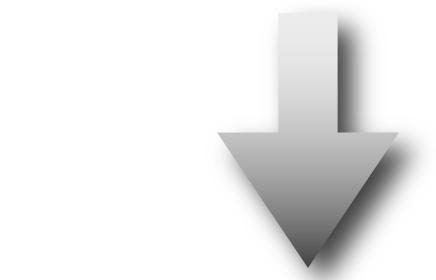
*“when should I consider breaking apart a service?”*



service  
functionality



code  
volatility

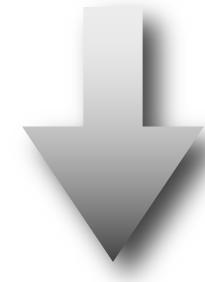


scalability and  
throughput

# service granularity

granularity disintegrators

*“when should I consider breaking apart a service?”*



scalability and  
throughput

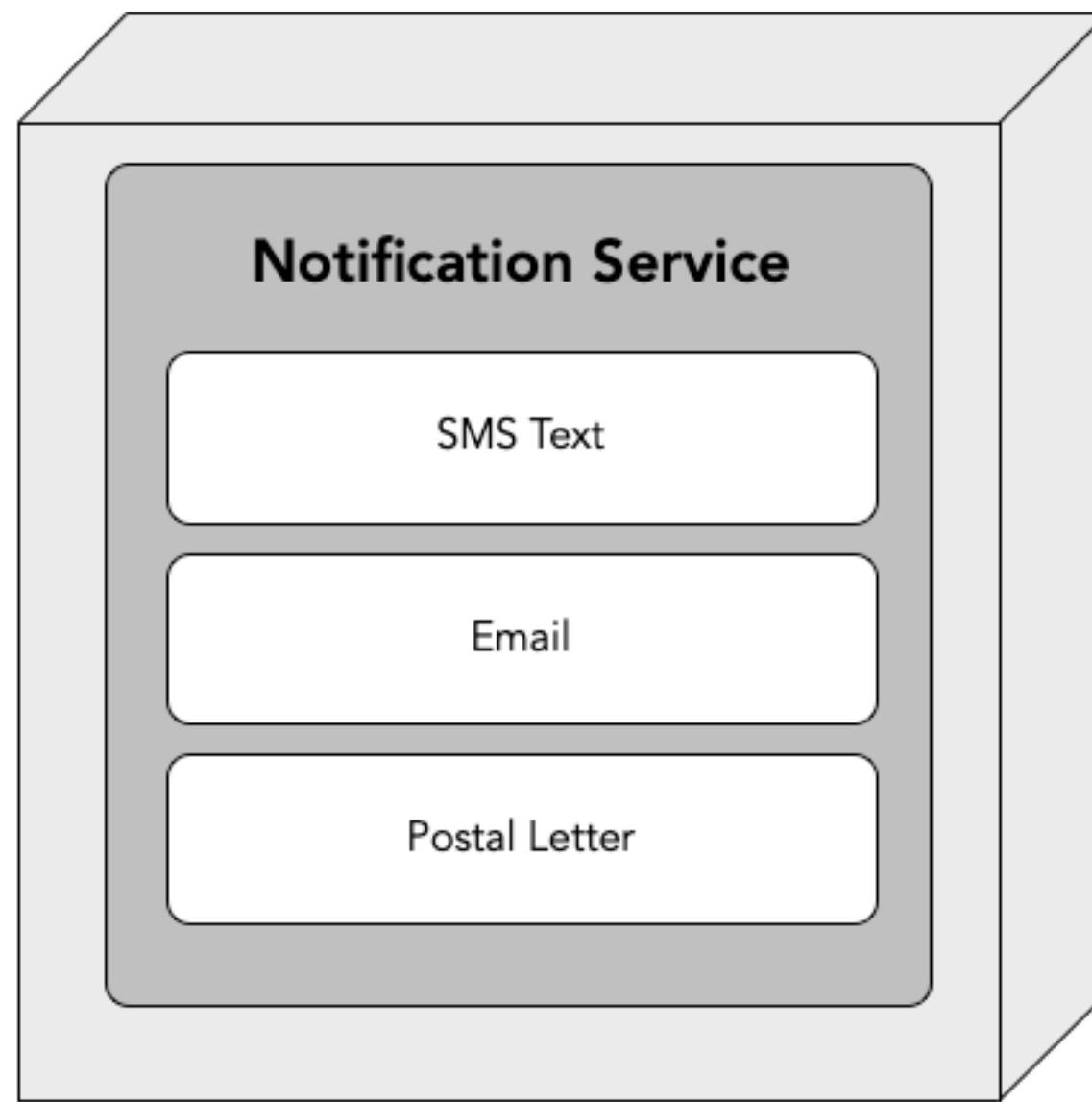
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



scalability and  
throughput



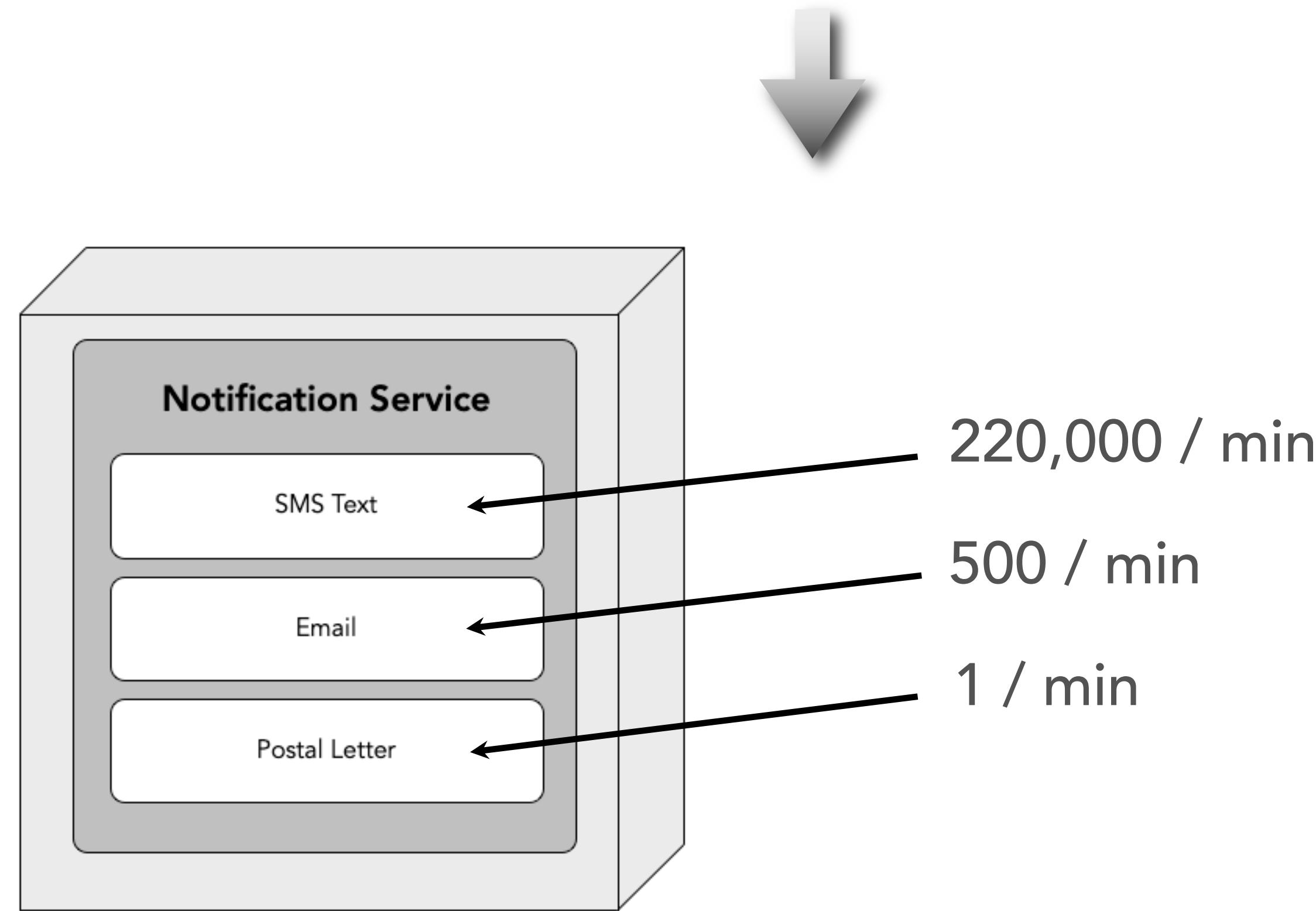
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



scalability and  
throughput



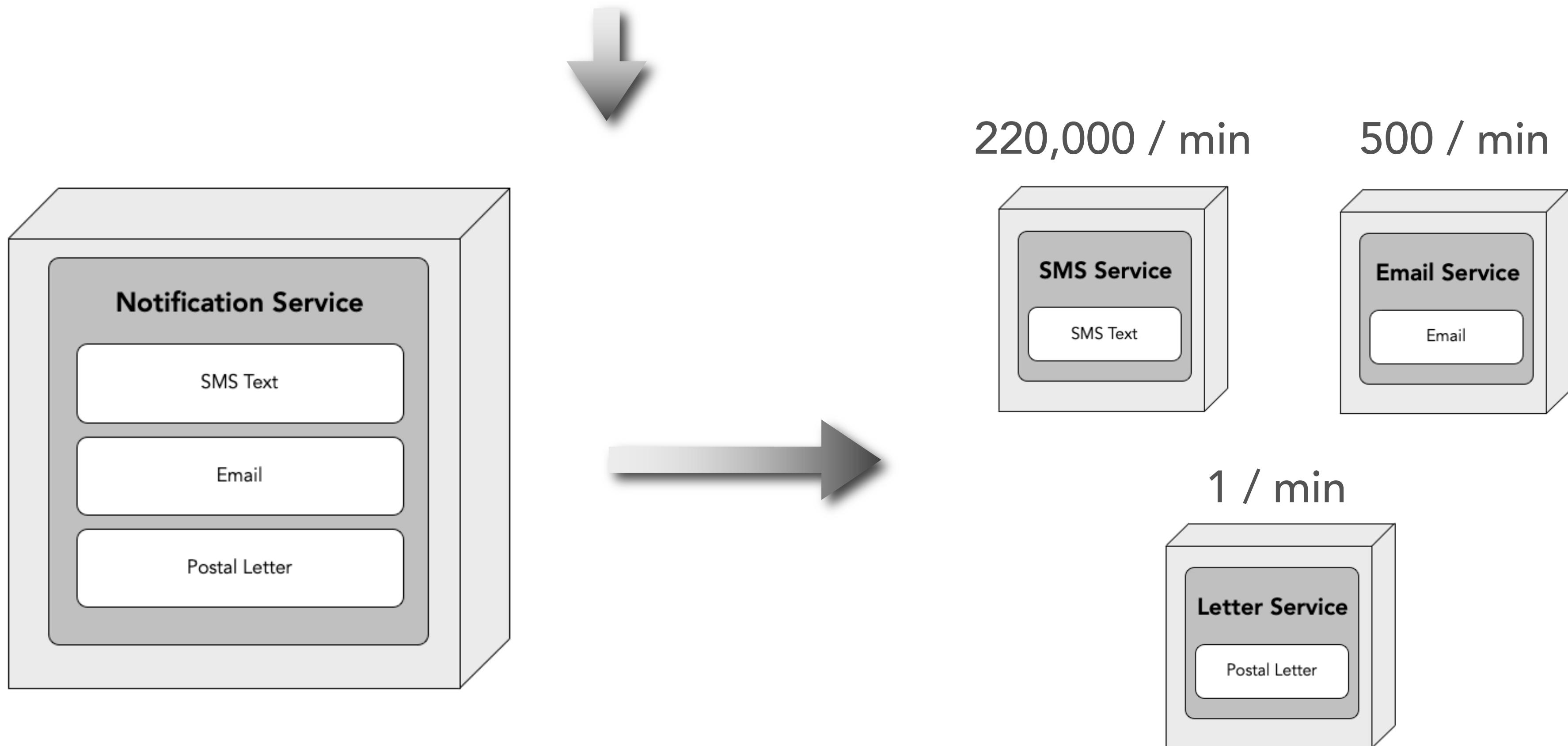
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



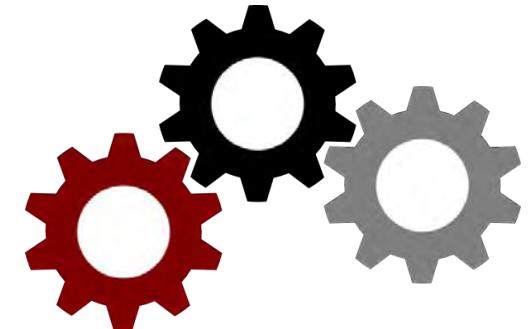
scalability and  
throughput



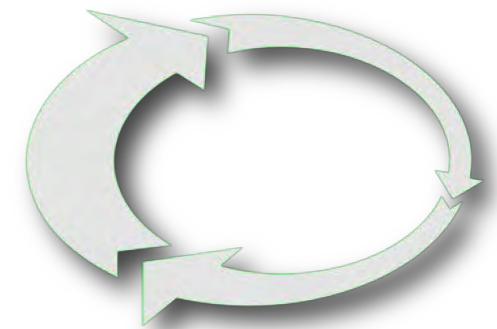
# service granularity

## granularity disintegrators

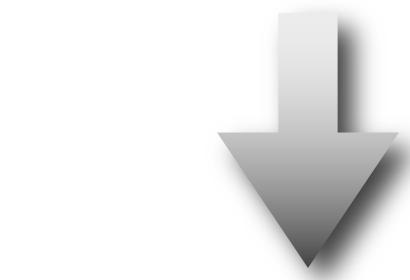
*“when should I consider breaking apart a service?”*



service  
functionality



code  
volatility

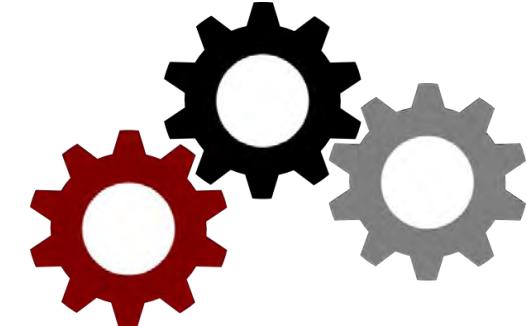


scalability and  
throughput

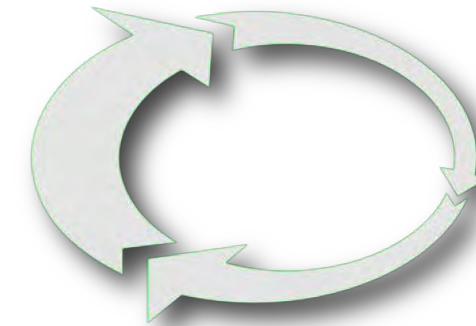
# service granularity

## granularity disintegrators

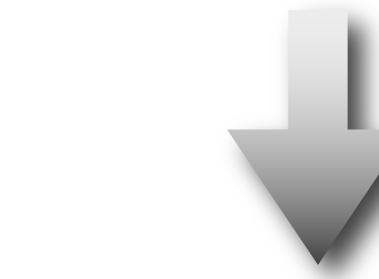
*“when should I consider breaking apart a service?”*



service  
functionality



code  
volatility



scalability and  
throughput

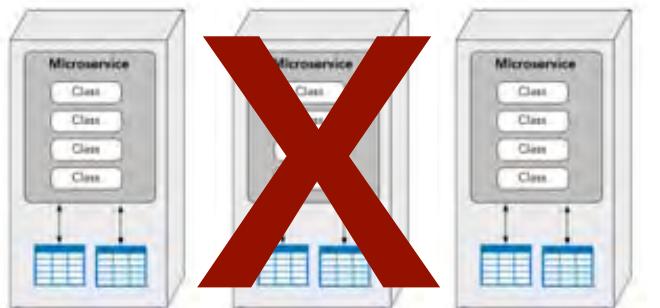
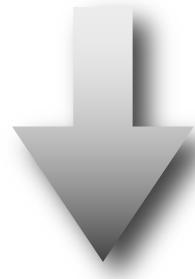


fault  
tolerance

# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*

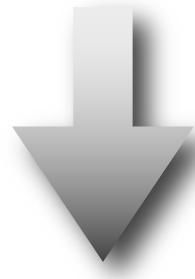


fault  
tolerance

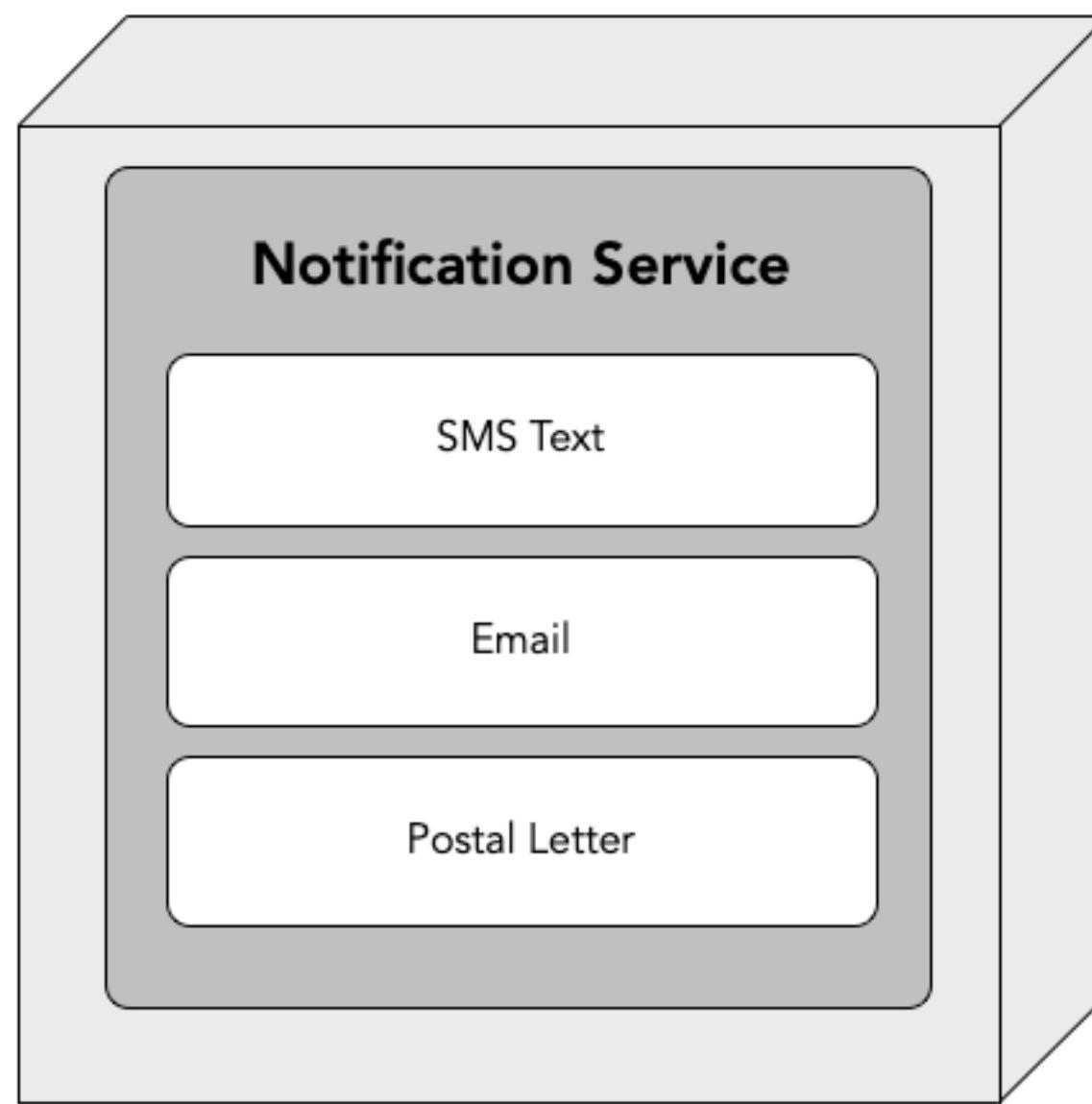
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



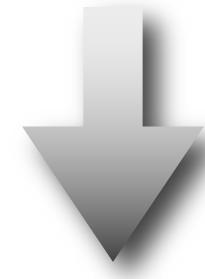
fault  
tolerance



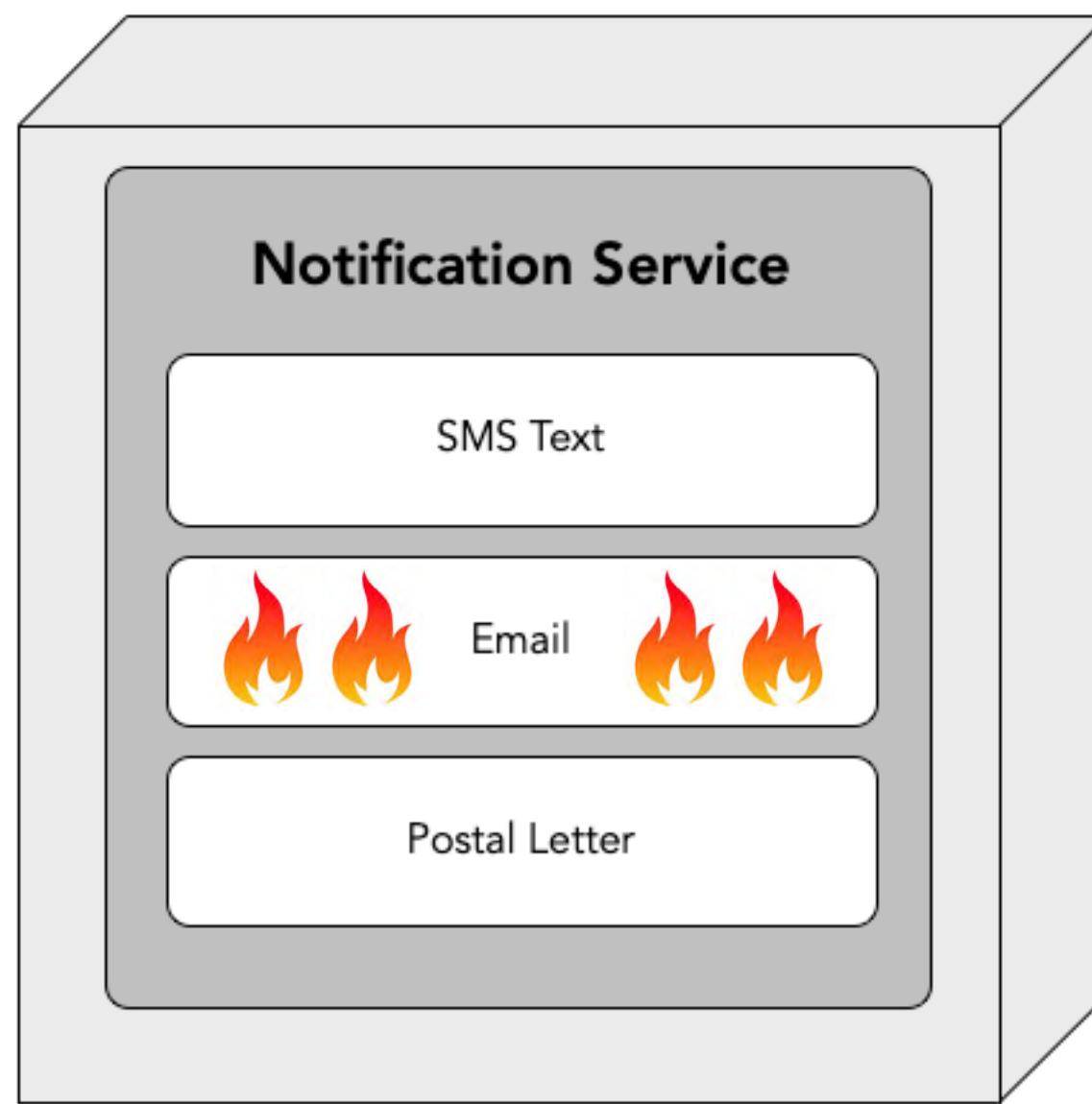
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



fault  
tolerance



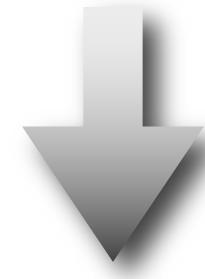
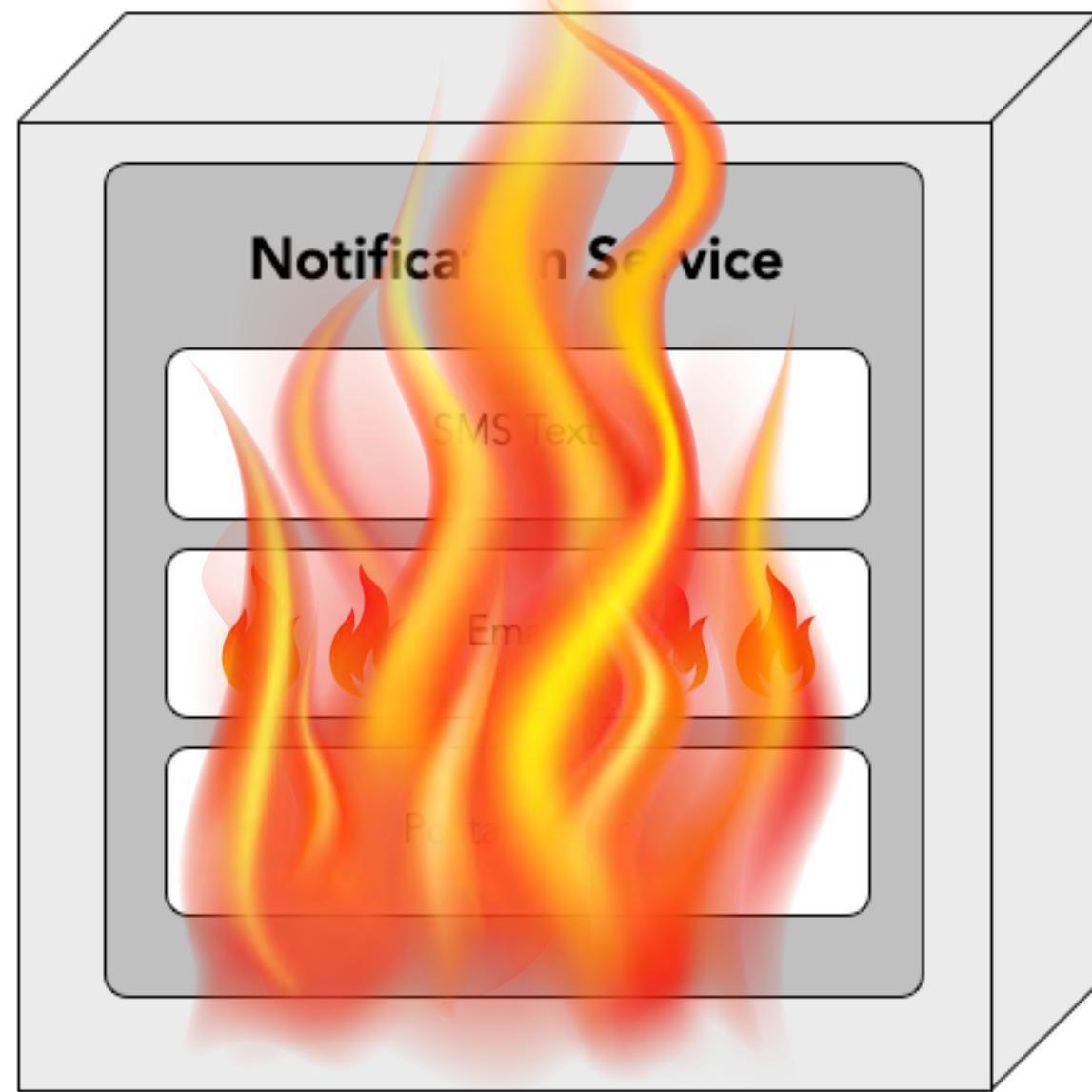
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



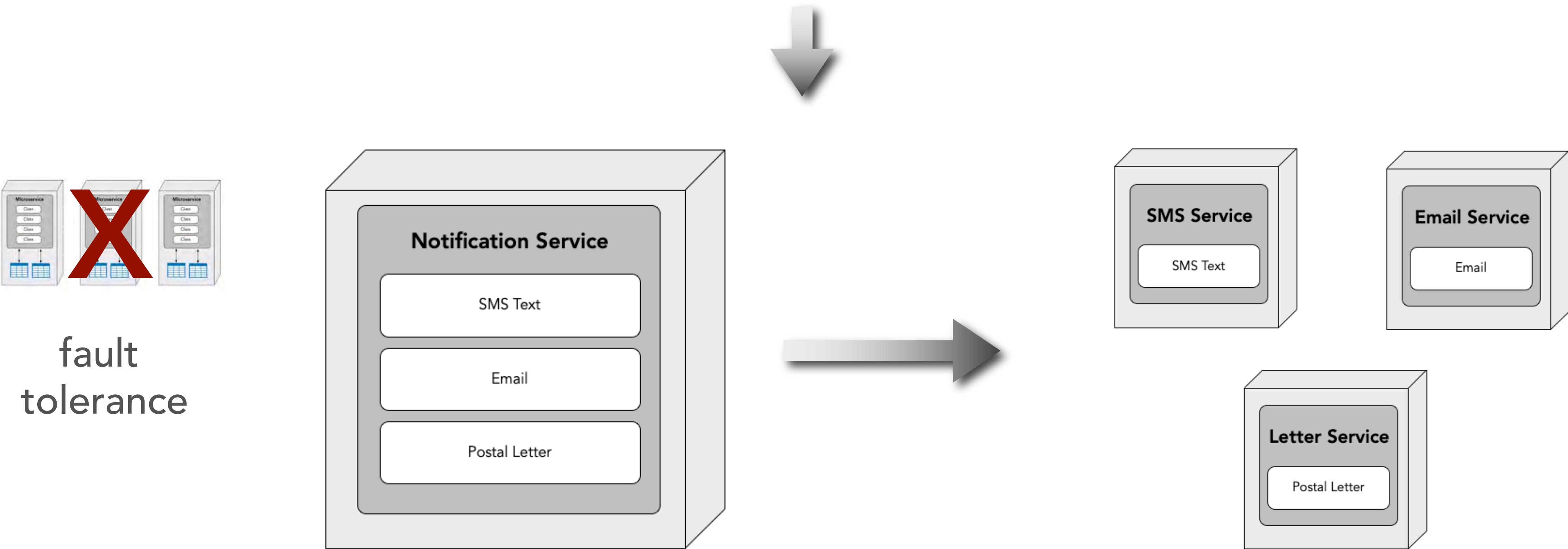
fault  
tolerance



# service granularity

## granularity disintegrators

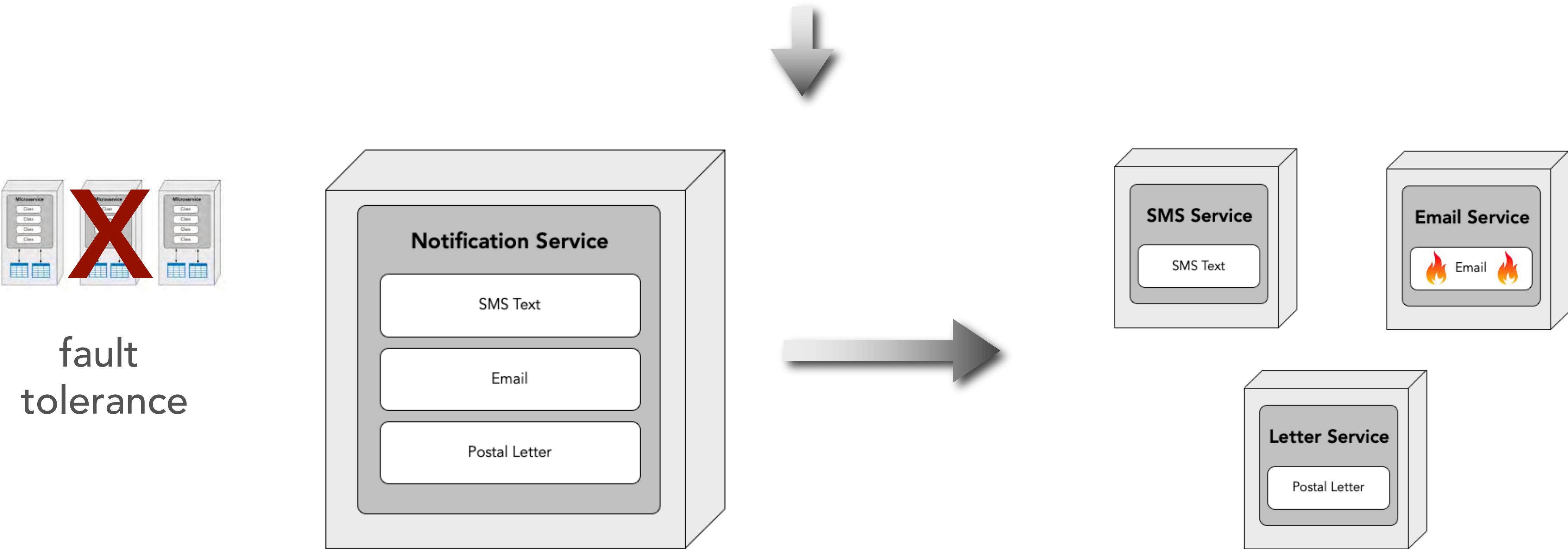
*“when should I consider breaking apart a service?”*



# service granularity

## granularity disintegrators

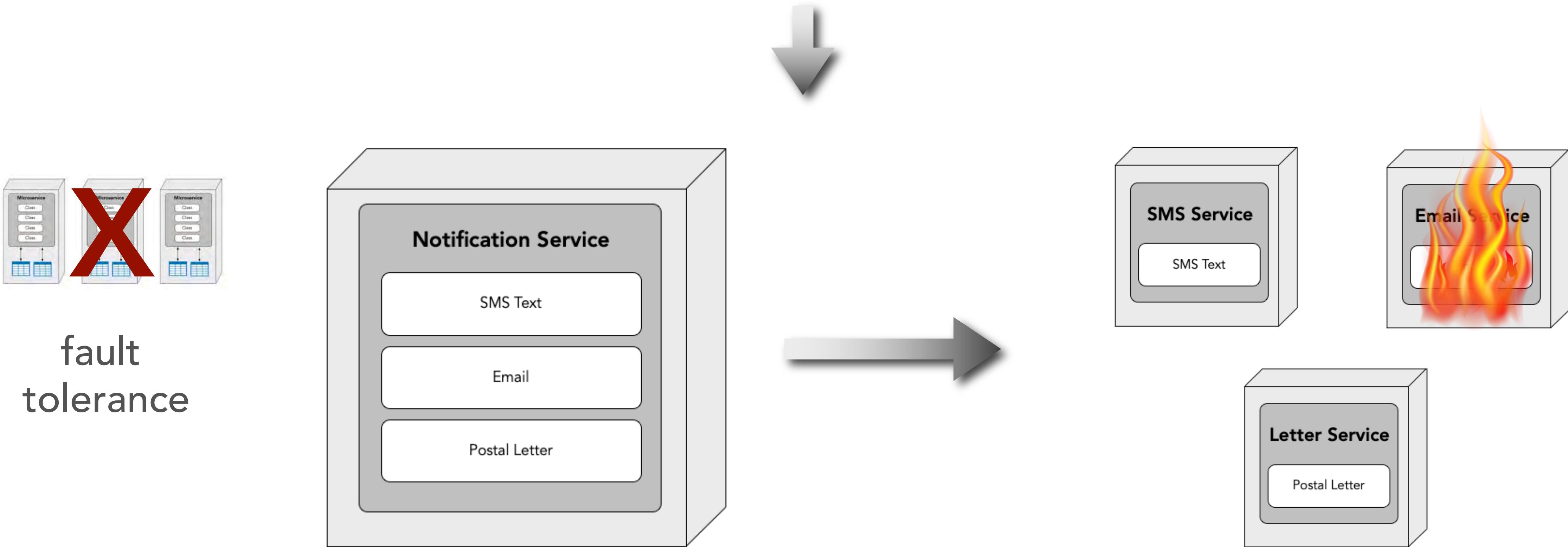
*“when should I consider breaking apart a service?”*



# service granularity

## granularity disintegrators

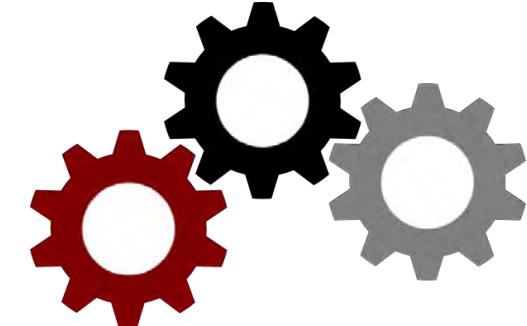
*“when should I consider breaking apart a service?”*



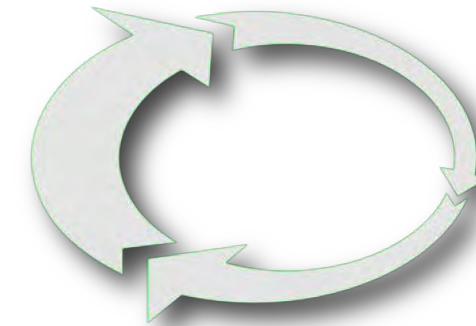
# service granularity

## granularity disintegrators

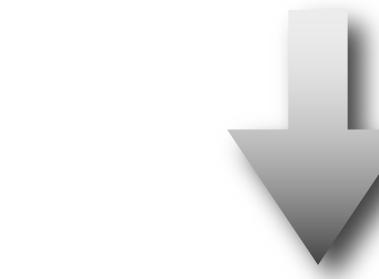
*“when should I consider breaking apart a service?”*



service  
functionality



code  
volatility



scalability and  
throughput

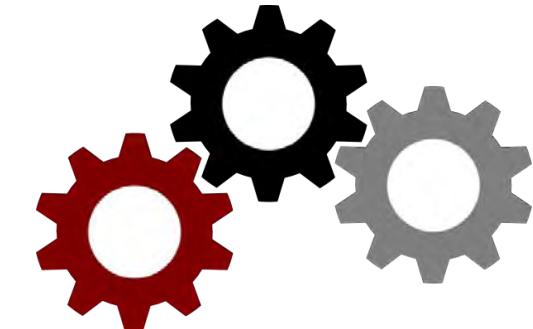


fault  
tolerance

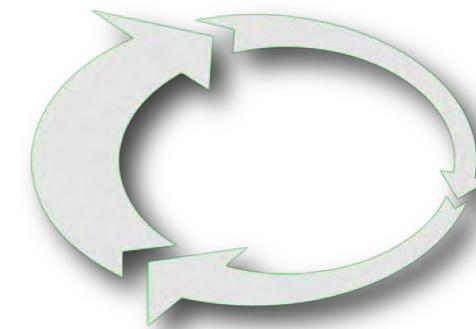
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



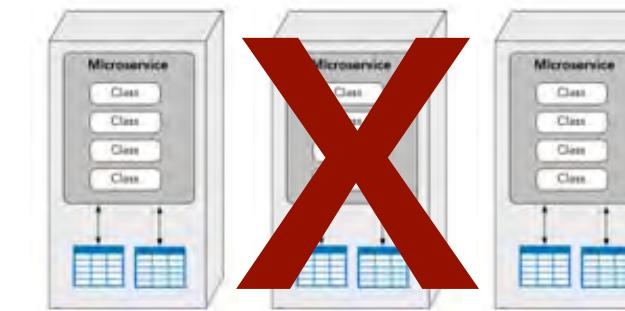
service  
functionality



code  
volatility



scalability and  
throughput



fault  
tolerance



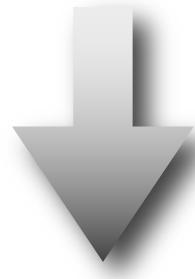
data  
security



# service granularity

granularity disintegrators

*“when should I consider breaking apart a service?”*

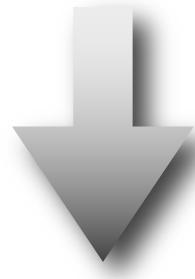


data  
security

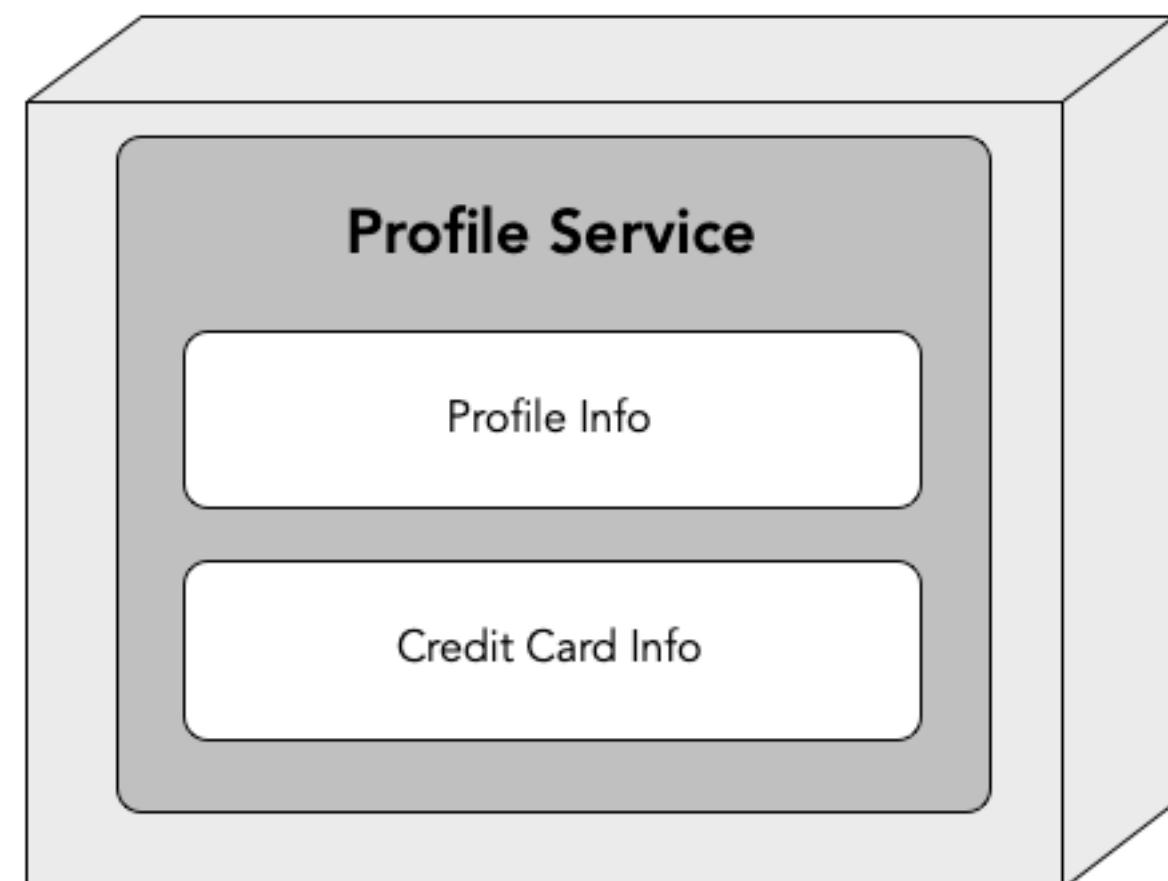
# service granularity

granularity disintegrators

*“when should I consider breaking apart a service?”*



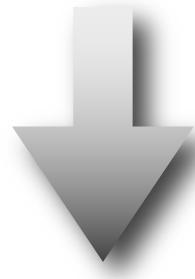
data  
security



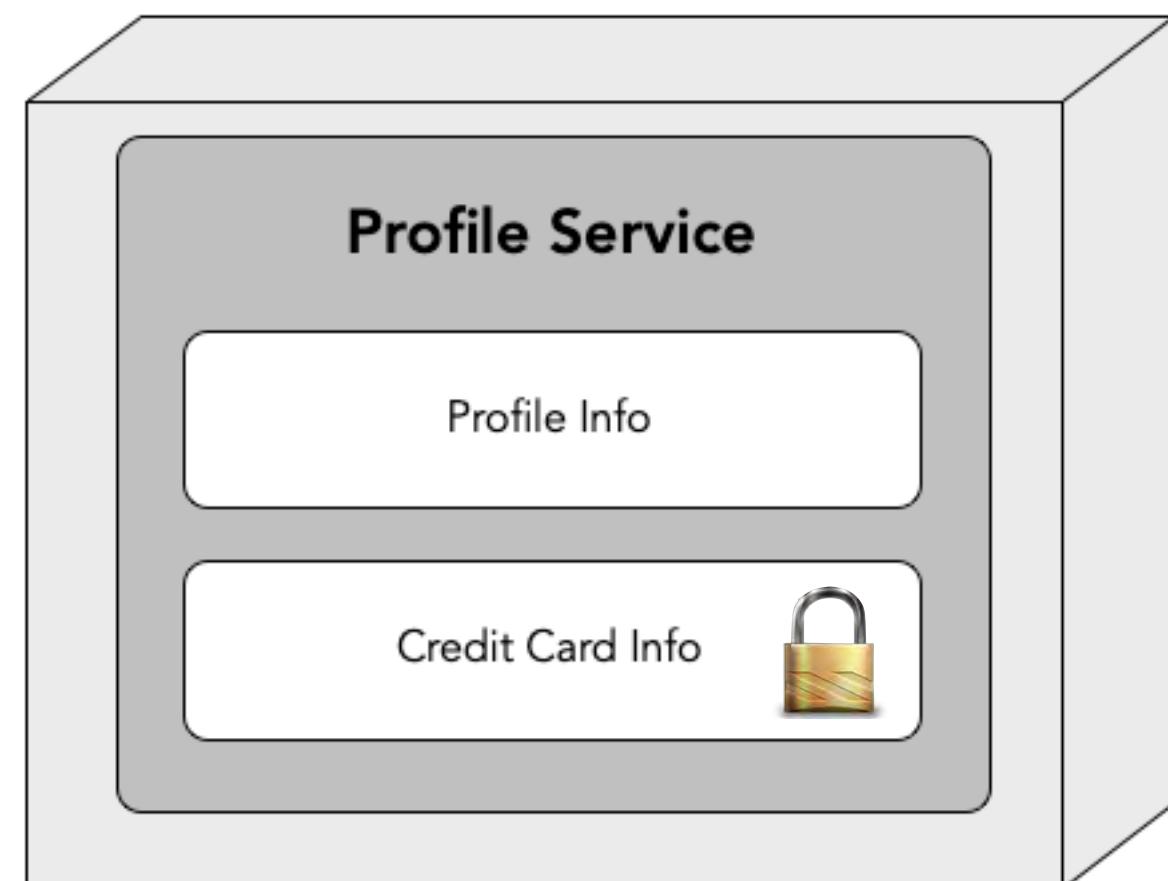
# service granularity

granularity disintegrators

*“when should I consider breaking apart a service?”*



data  
security



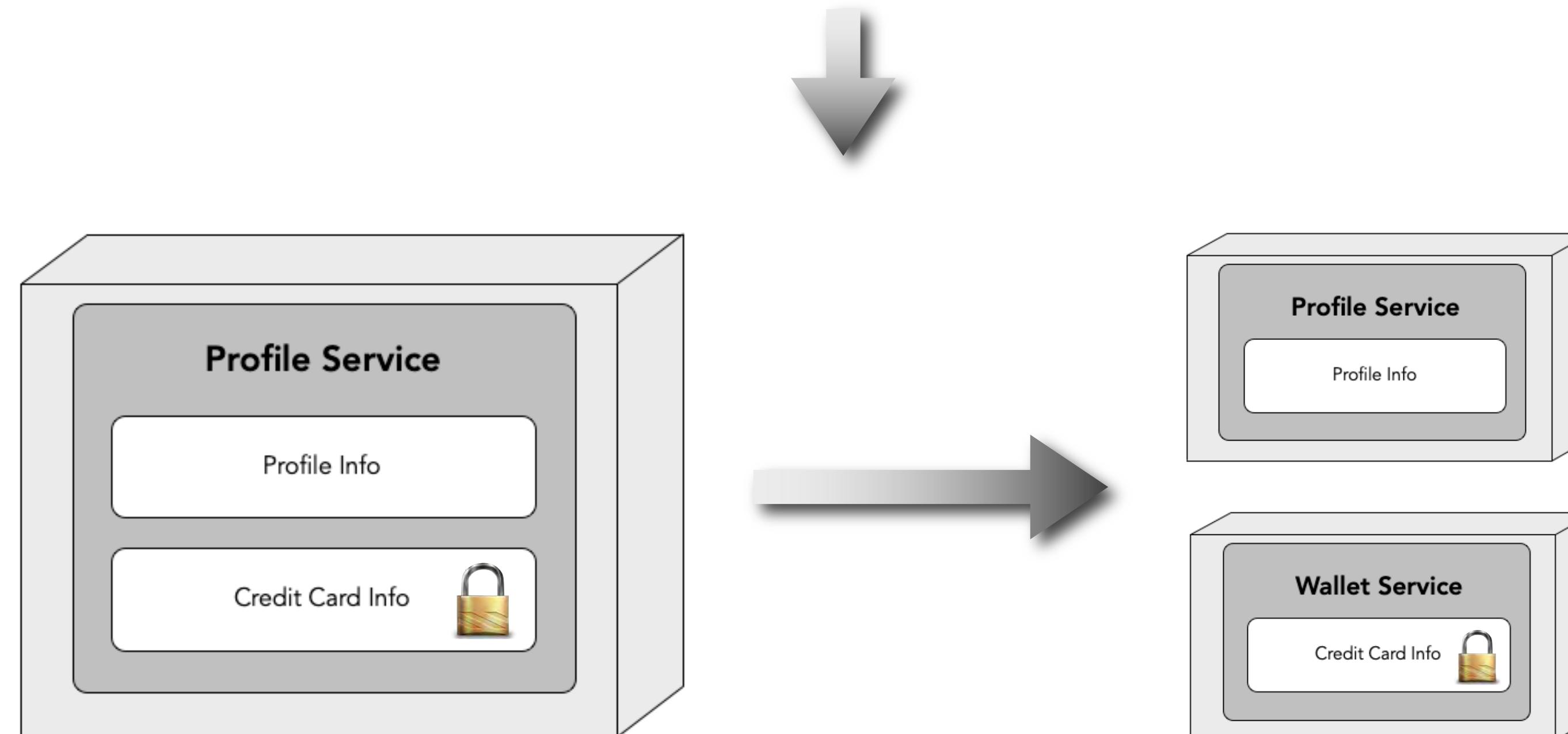
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



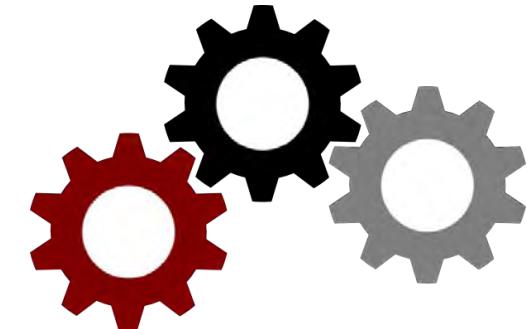
data  
security



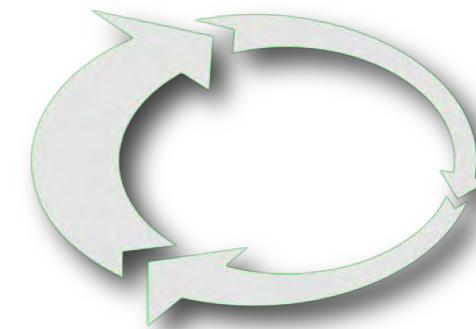
# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*



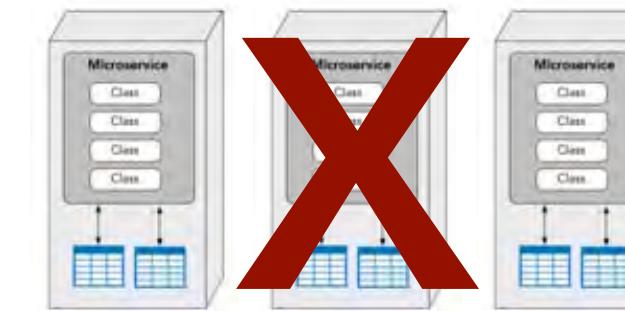
service  
functionality



code  
volatility



scalability and  
throughput



fault  
tolerance



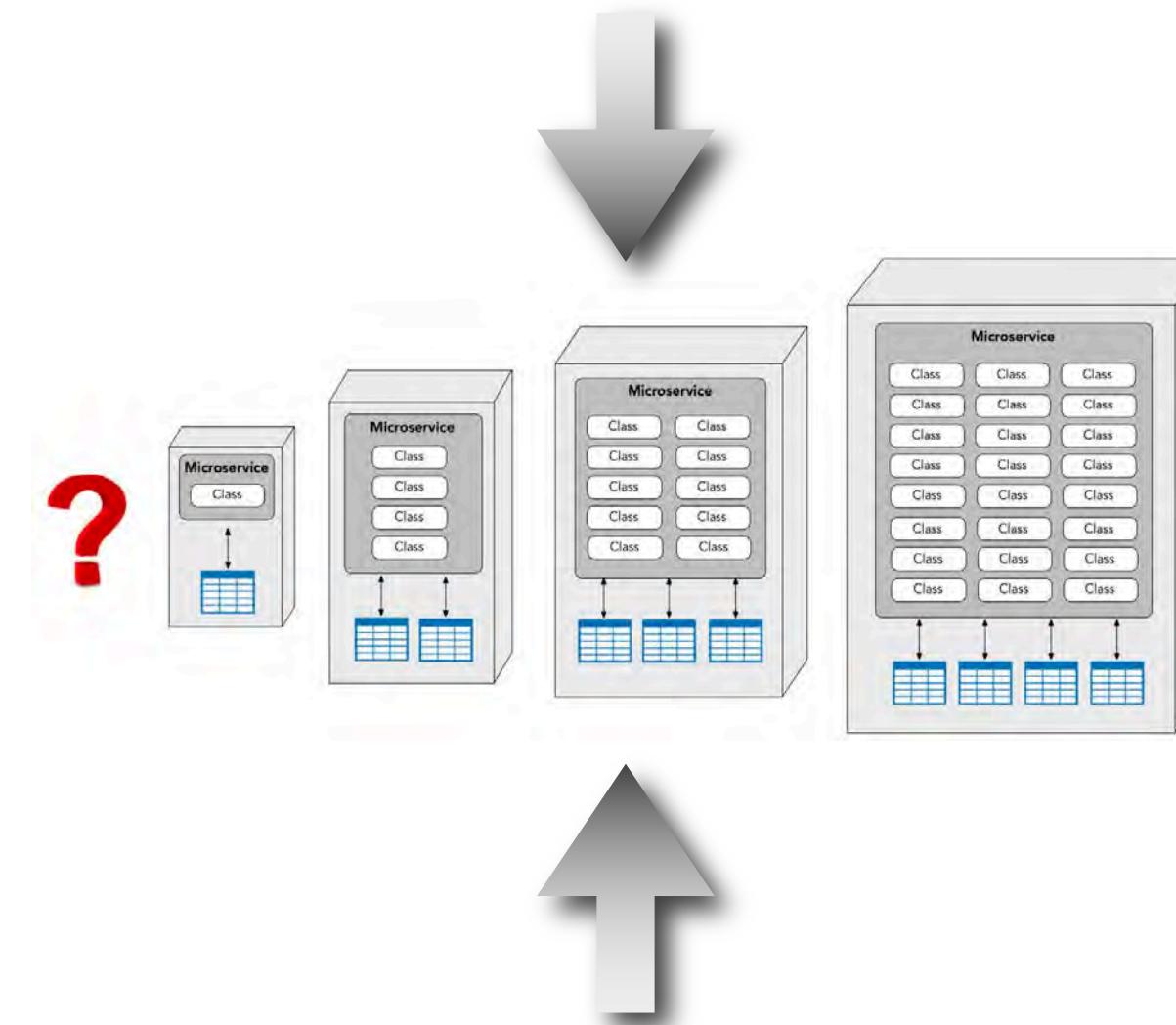
data  
security



# service granularity

## granularity disintegrators

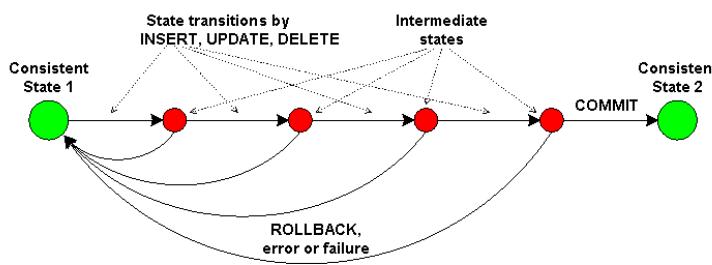
*“when should I consider breaking apart a service?”*



## granularity integrators

*“when should I consider putting services back together?”*

# service granularity

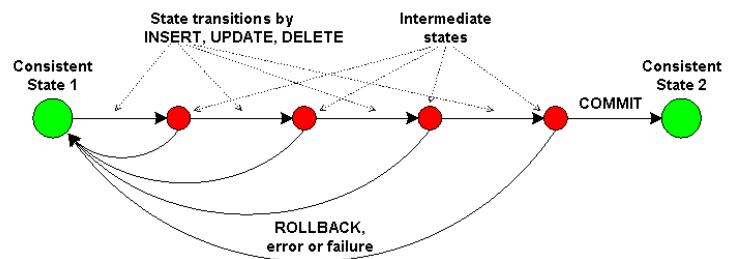


database  
transactions

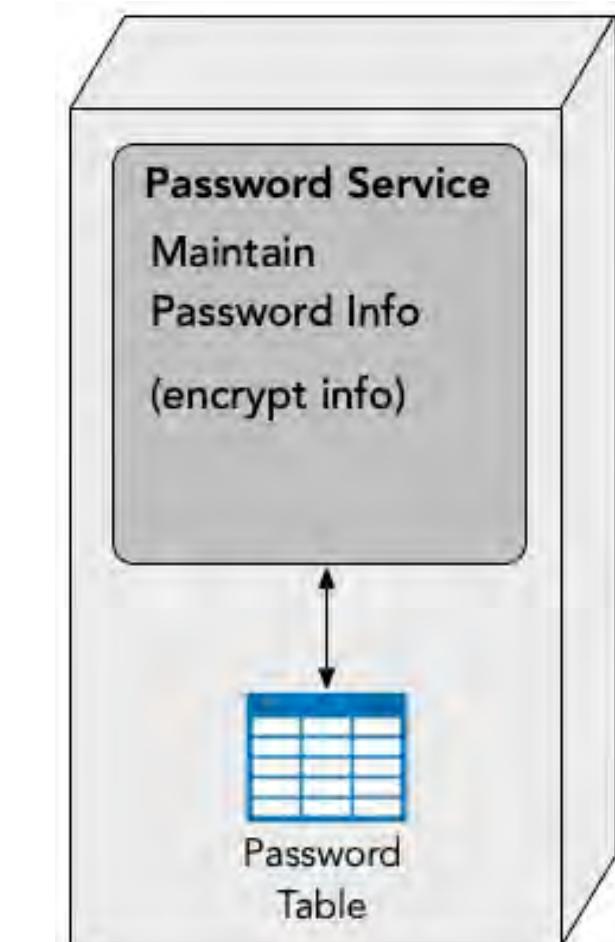
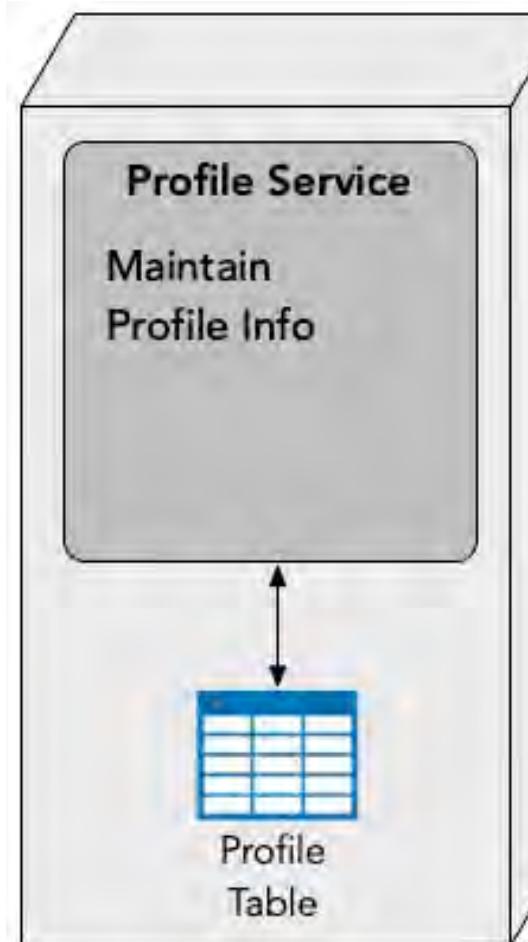


granularity integrators  
*“when should I consider putting services back together?”*

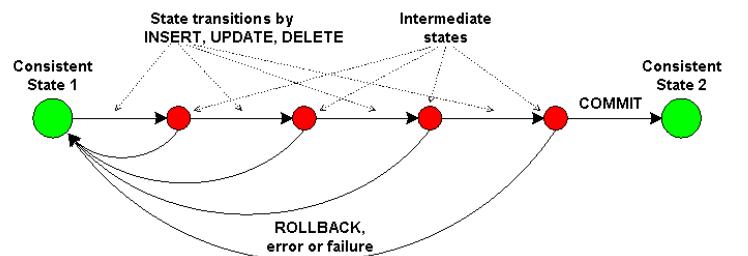
# service granularity



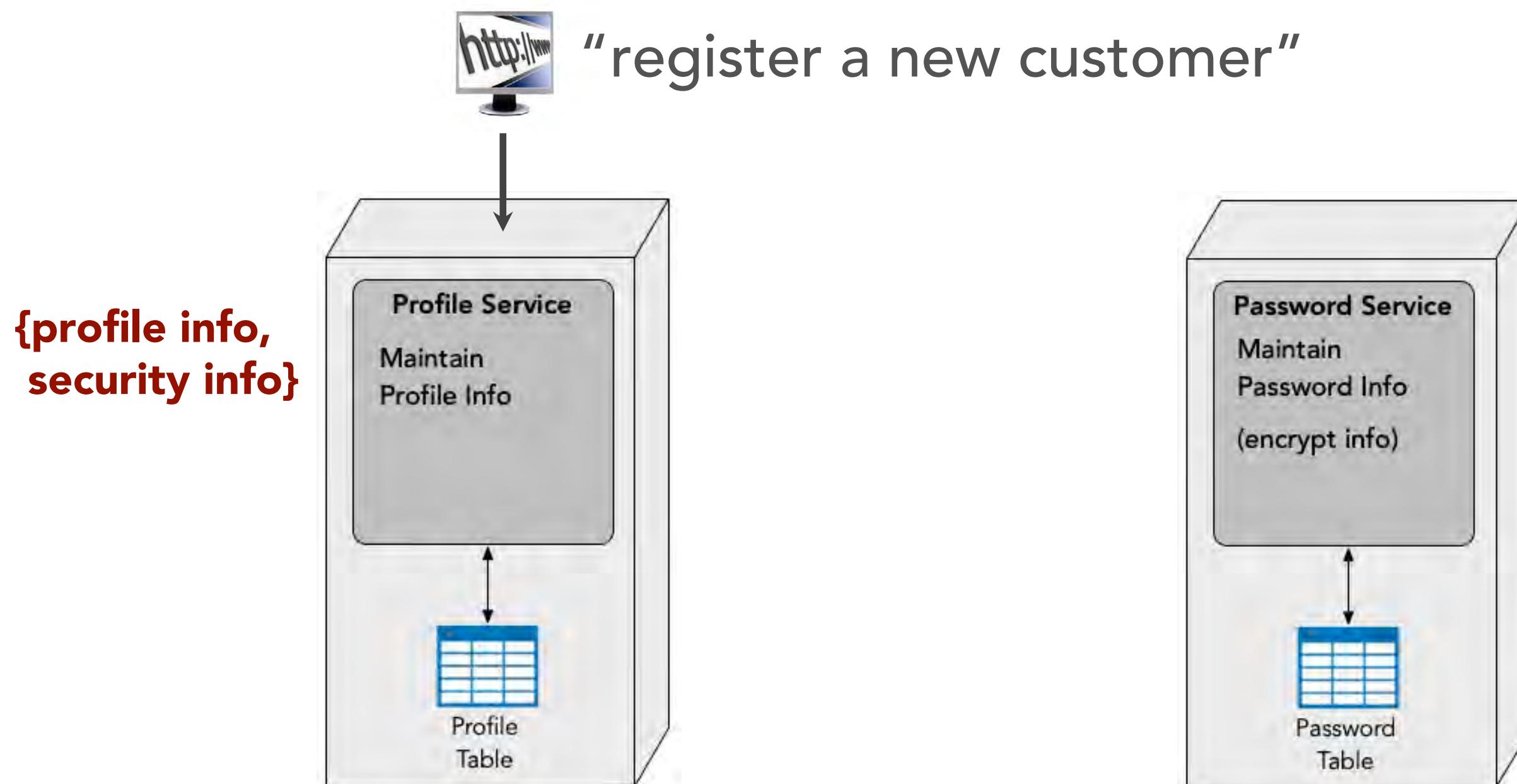
database  
transactions



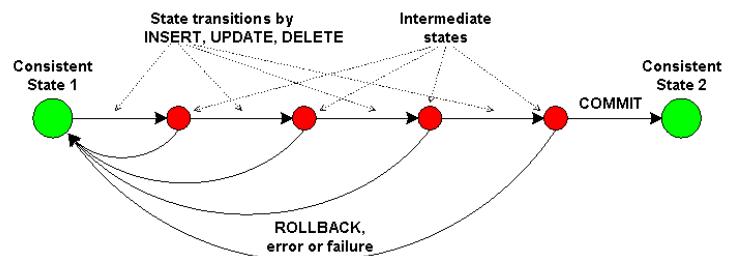
# service granularity



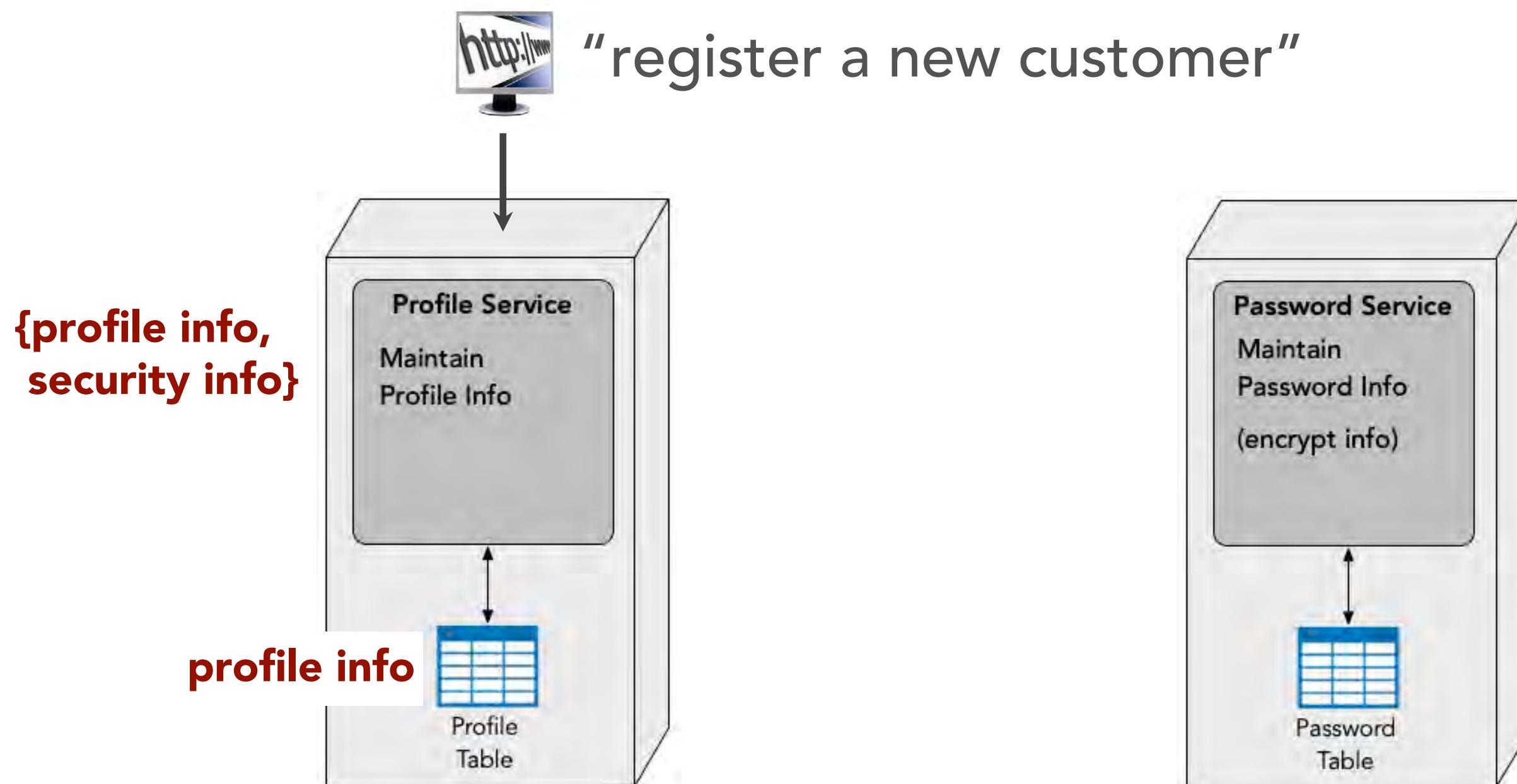
database  
transactions



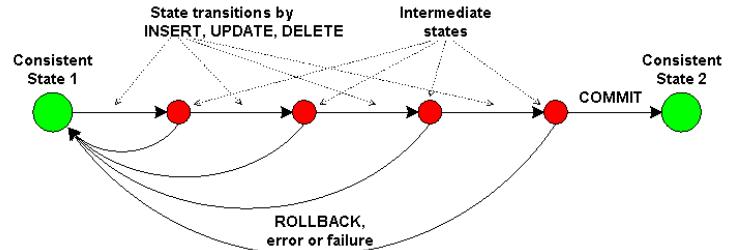
# service granularity



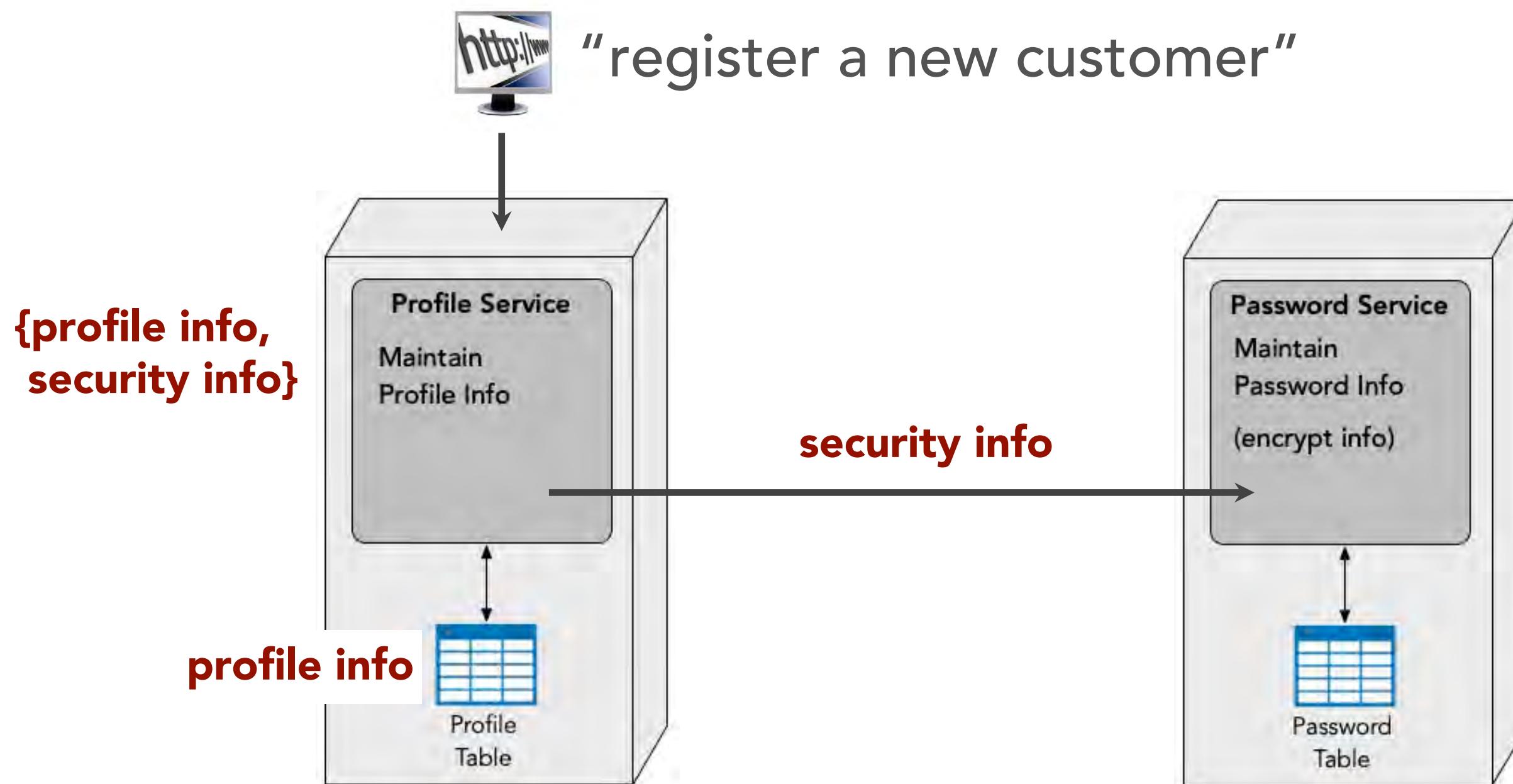
database  
transactions



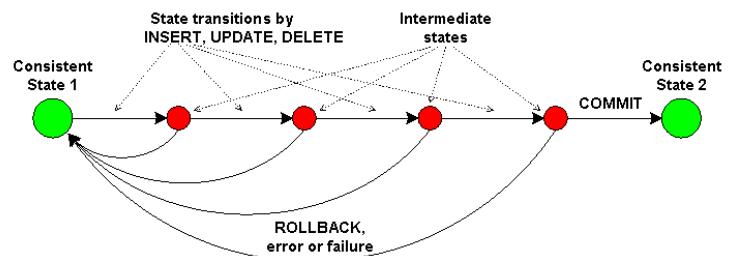
# service granularity



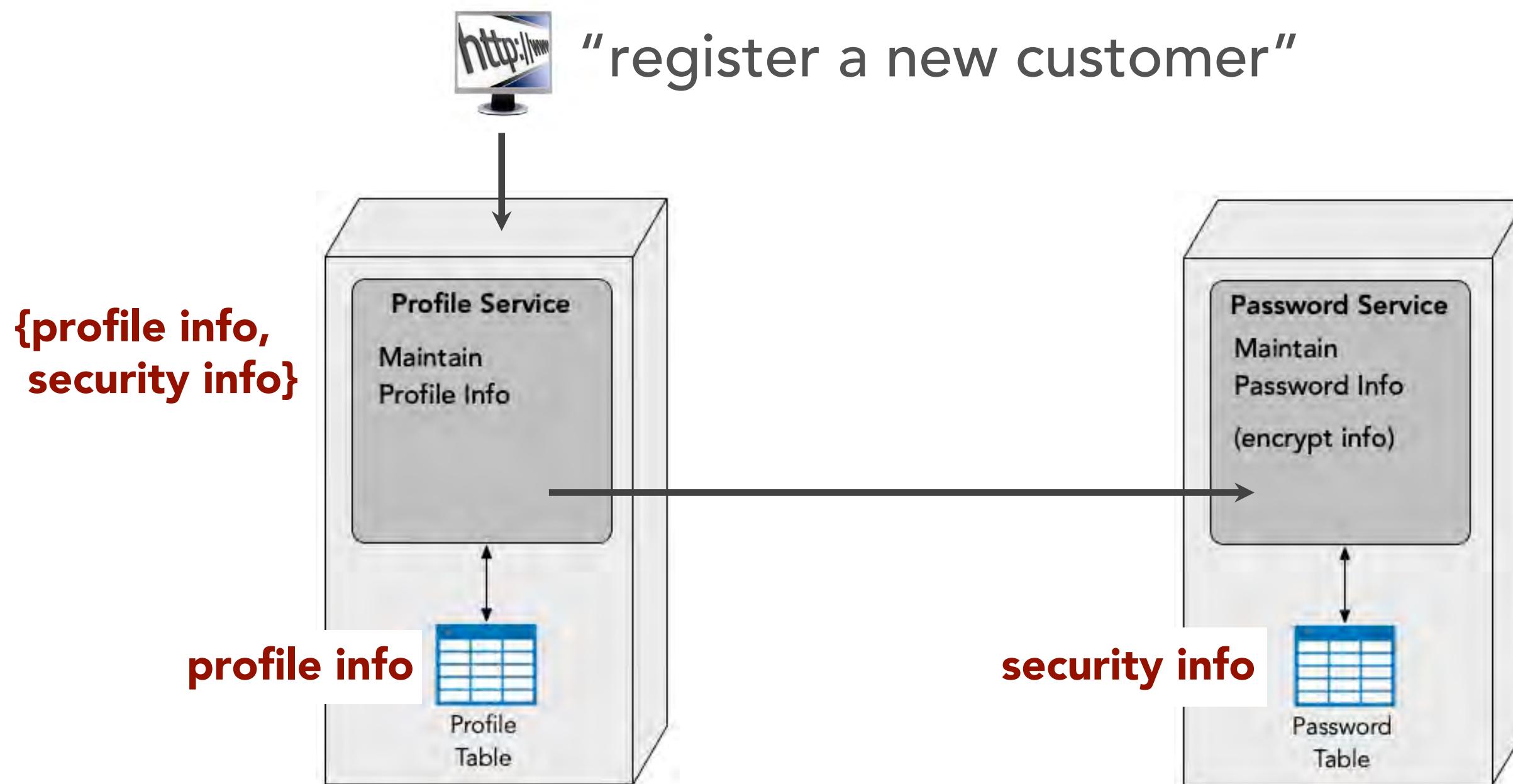
database  
transactions



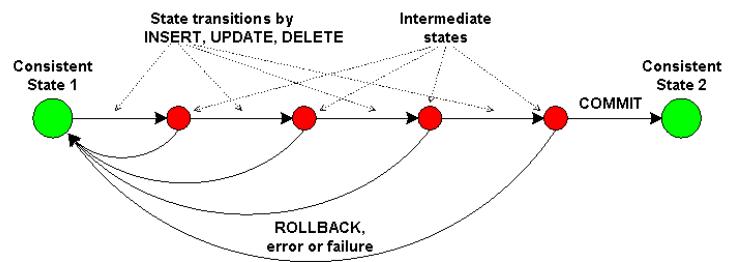
# service granularity



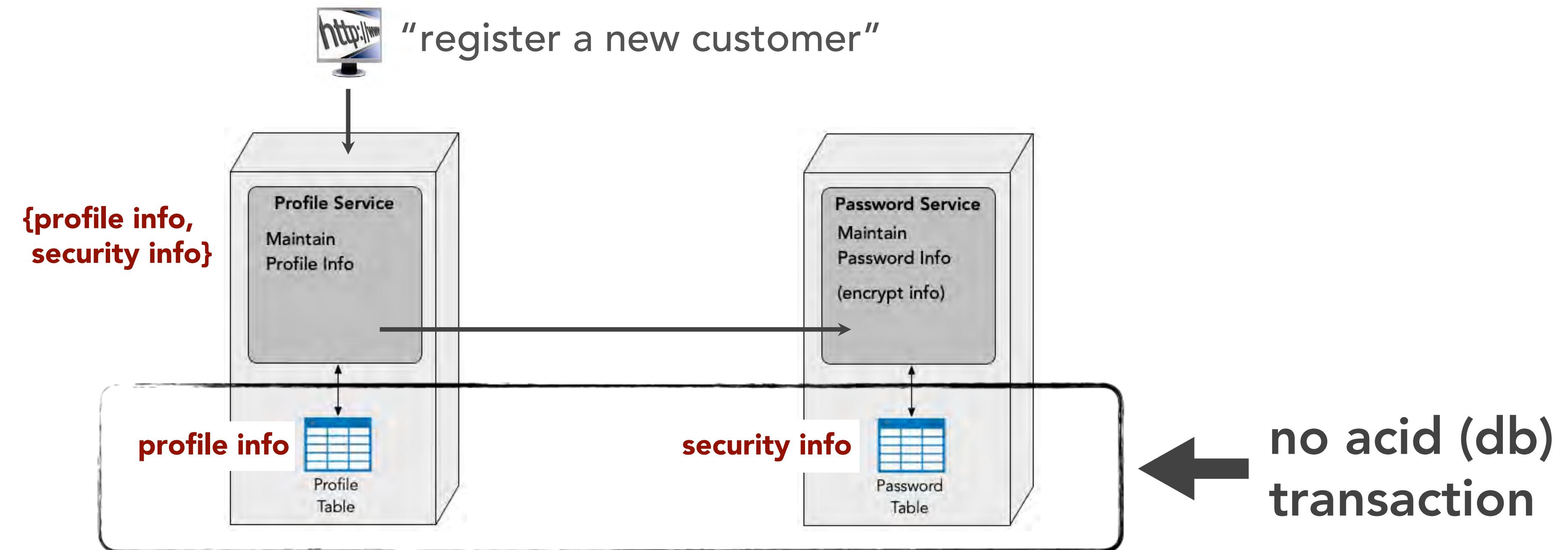
database  
transactions



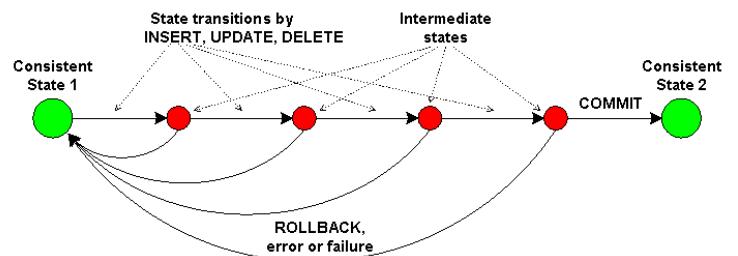
# service granularity



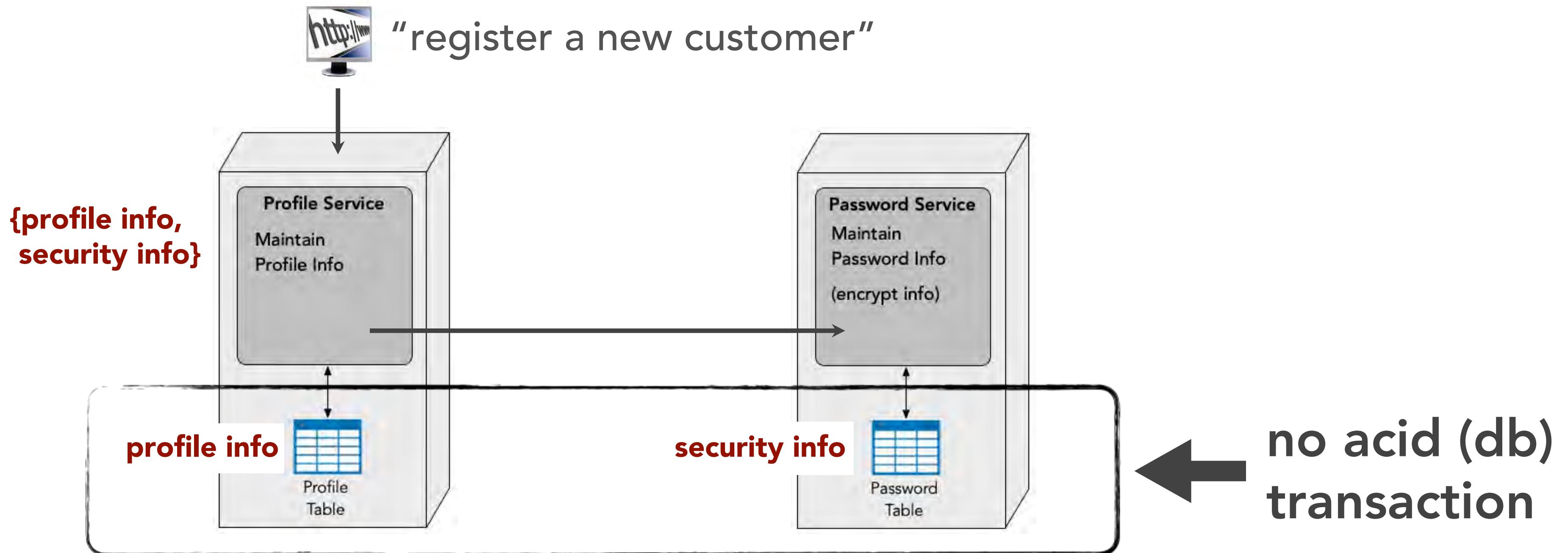
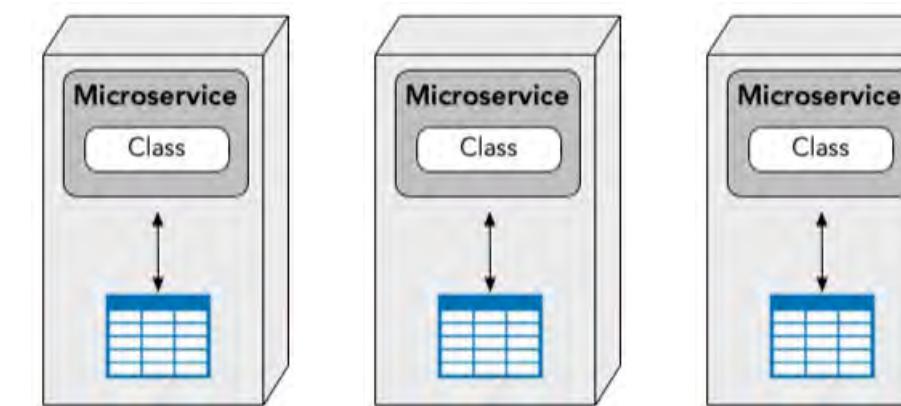
database  
transactions



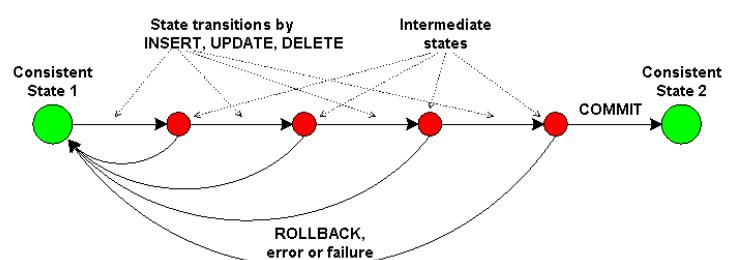
# service granularity



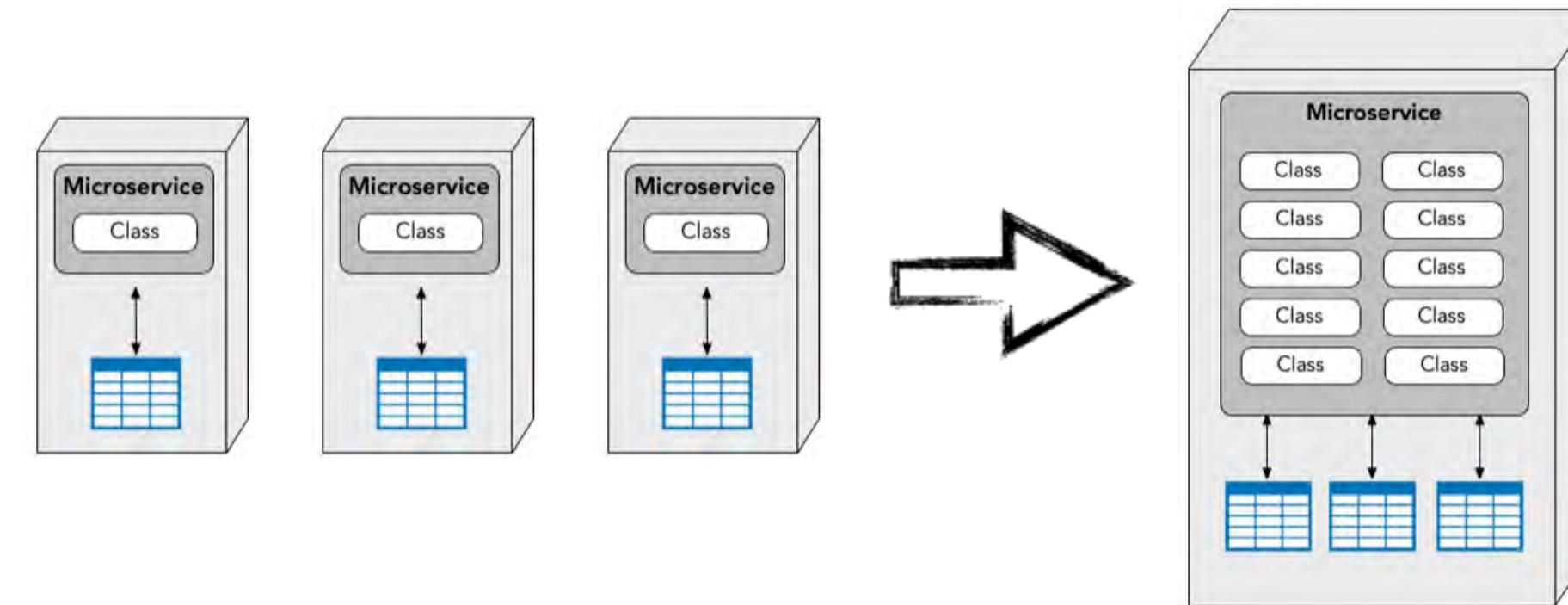
database  
transactions



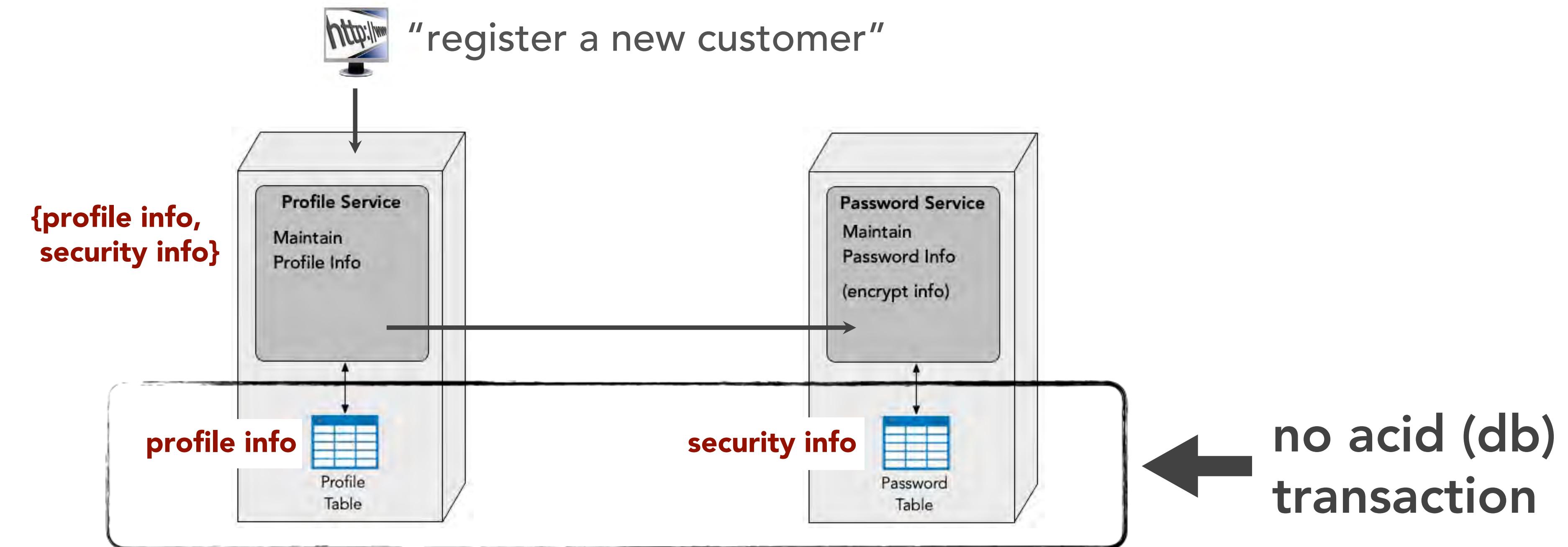
# service granularity



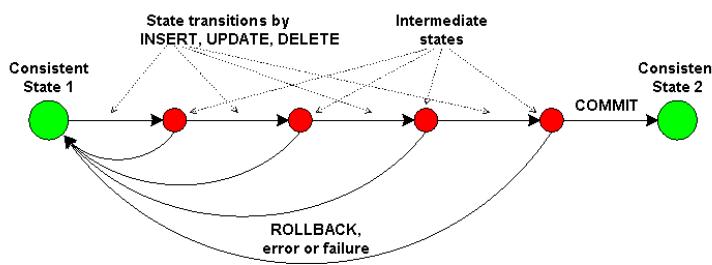
database  
transactions



"register a new customer"



# service granularity

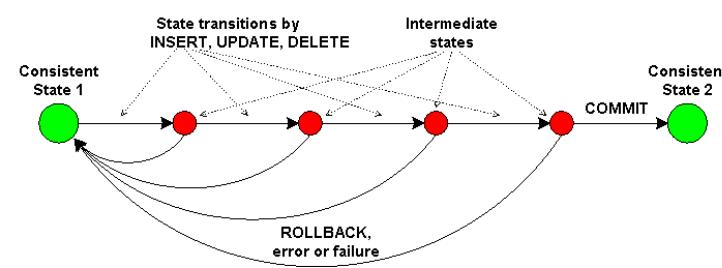


database  
transactions



granularity integrators  
*“when should I consider putting services back together?”*

# service granularity



database  
transactions

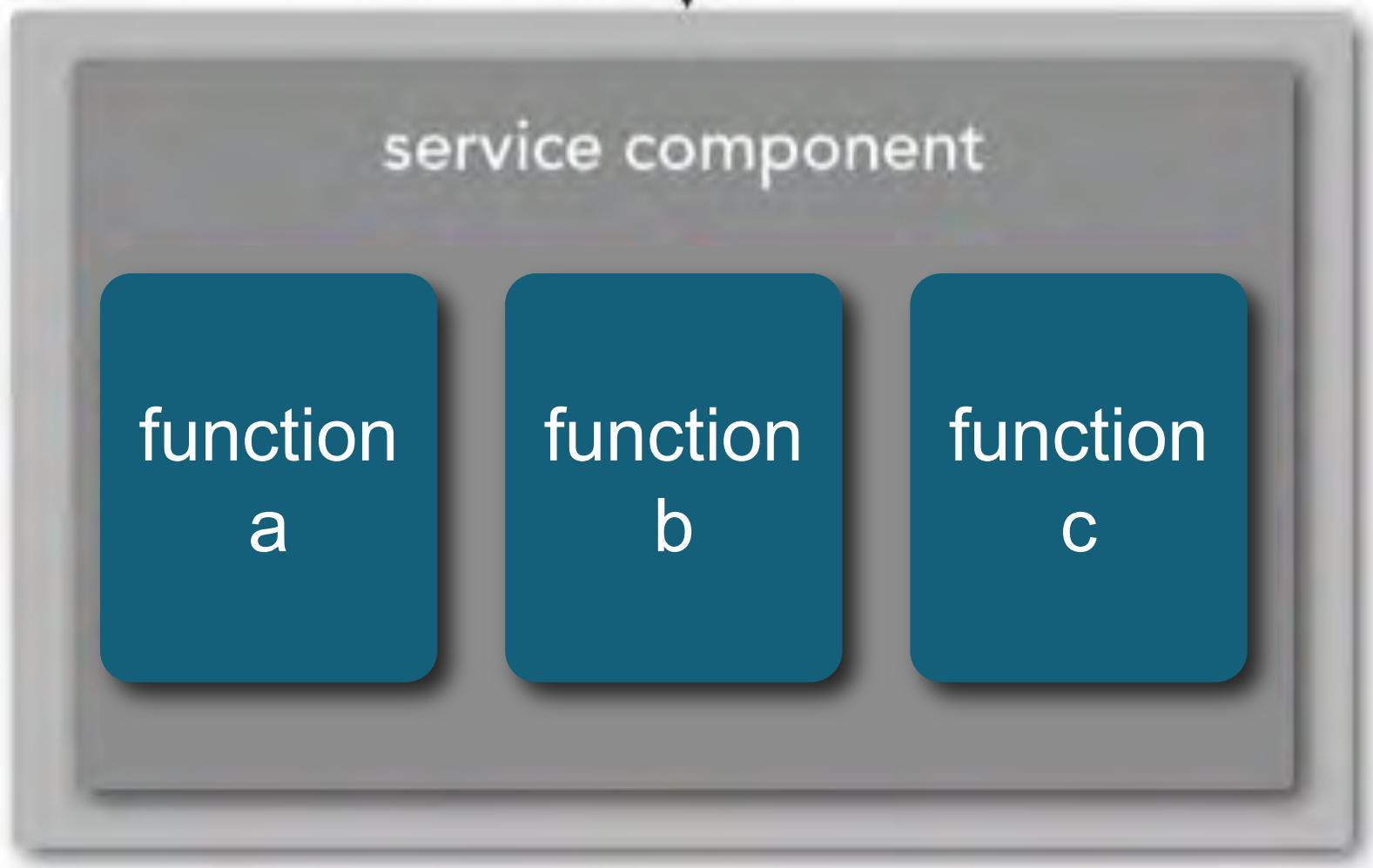
data  
dependencies



granularity integrators

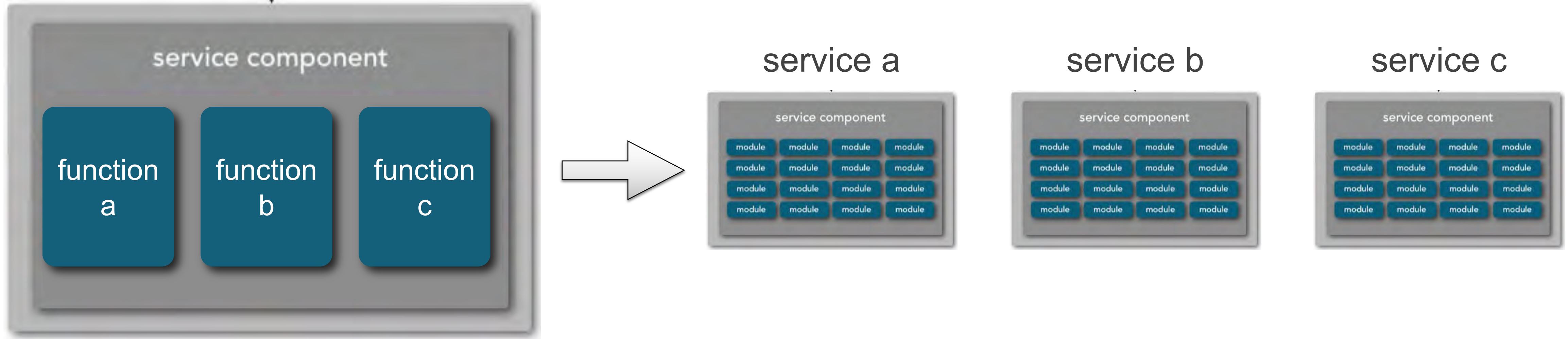
*“when should I consider putting services back together?”*

# service granularity



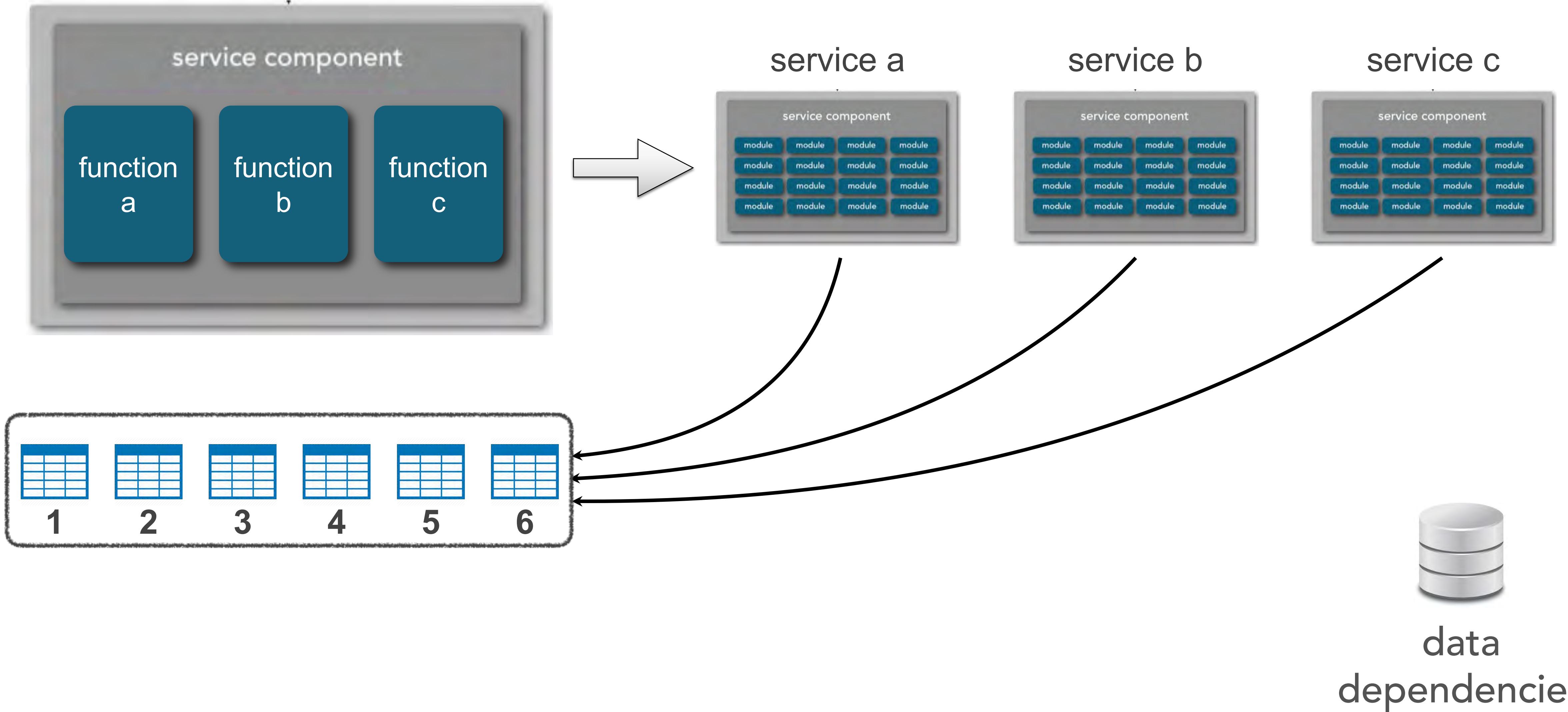
data  
dependencies

# service granularity

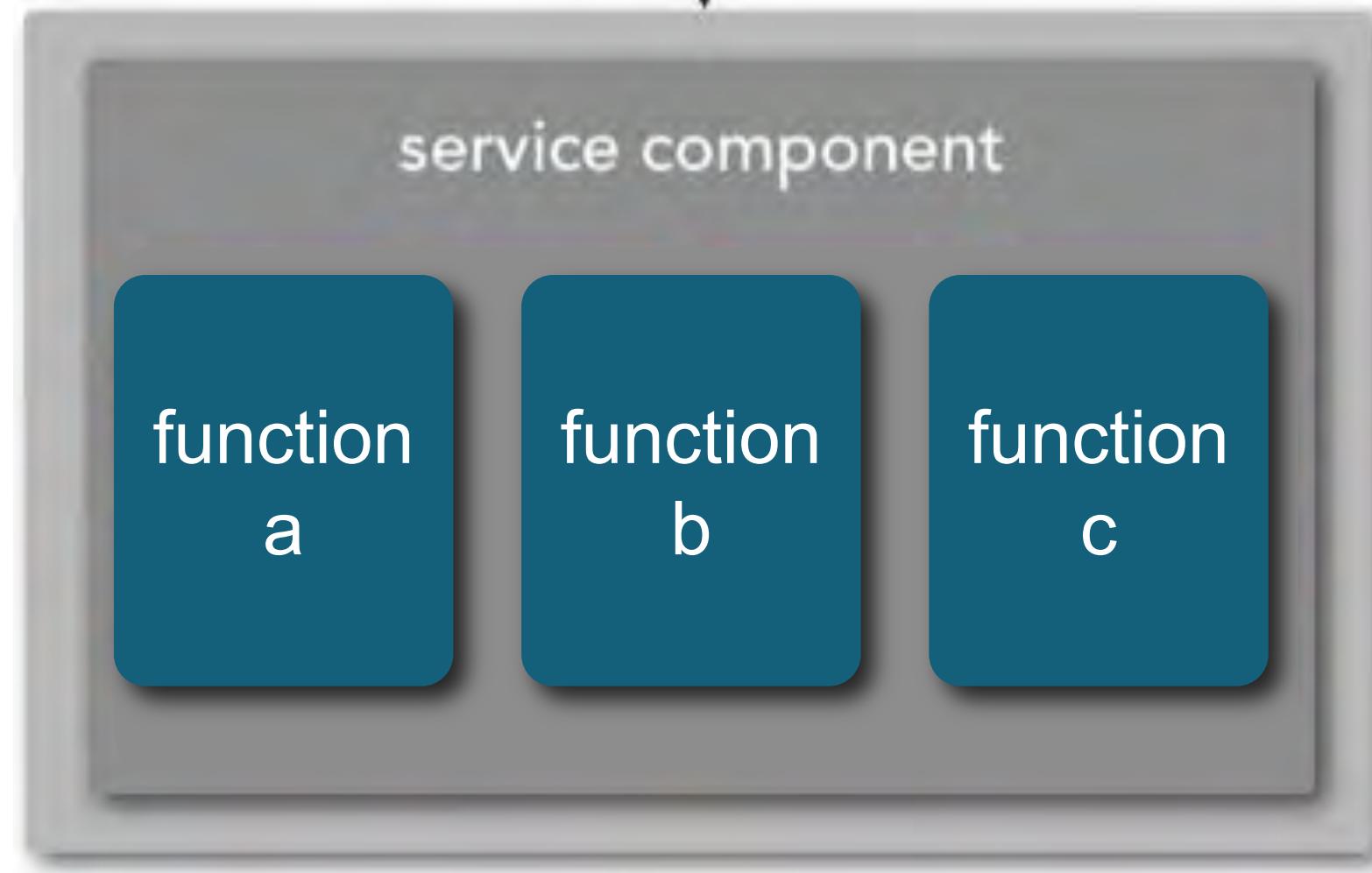


data  
dependencies

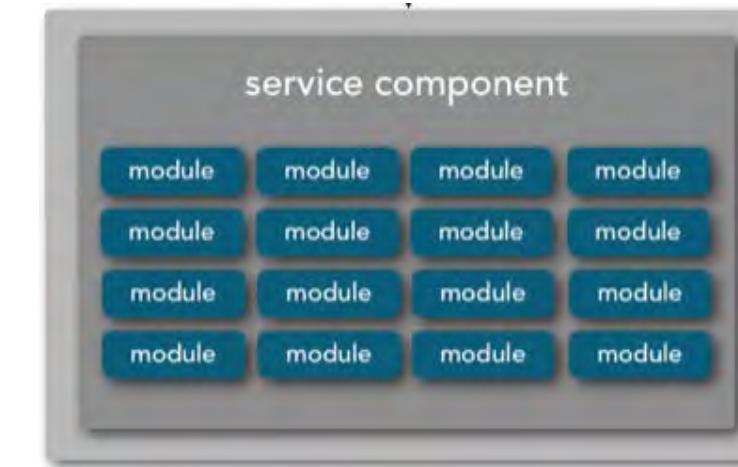
# service granularity



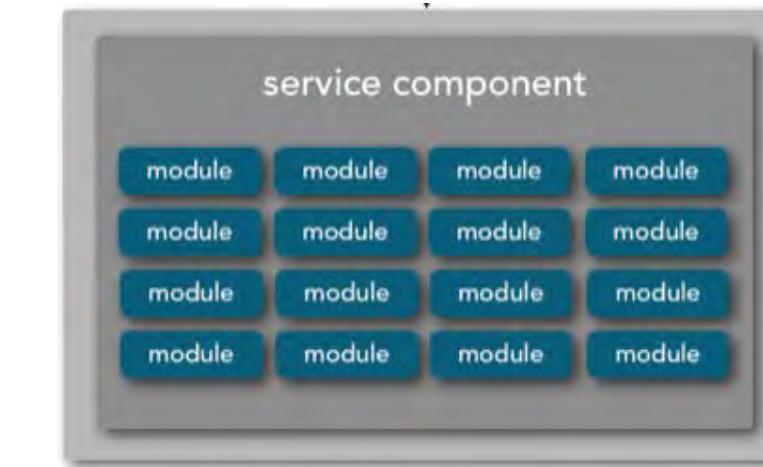
# service granularity



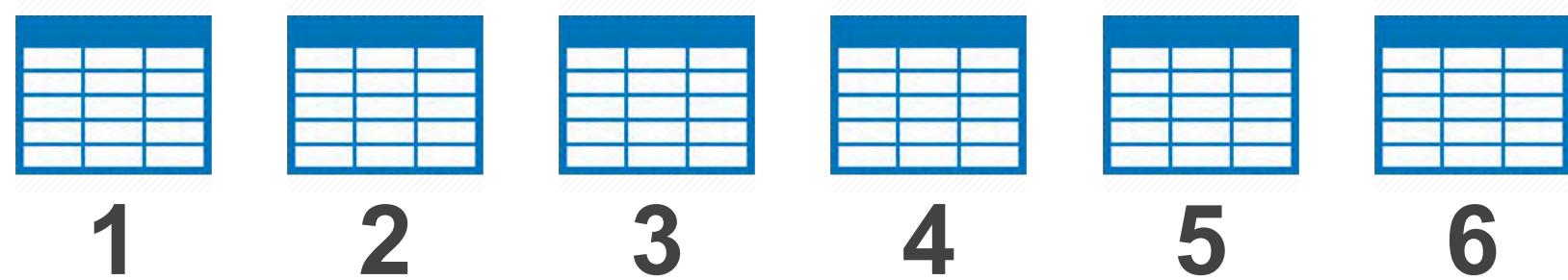
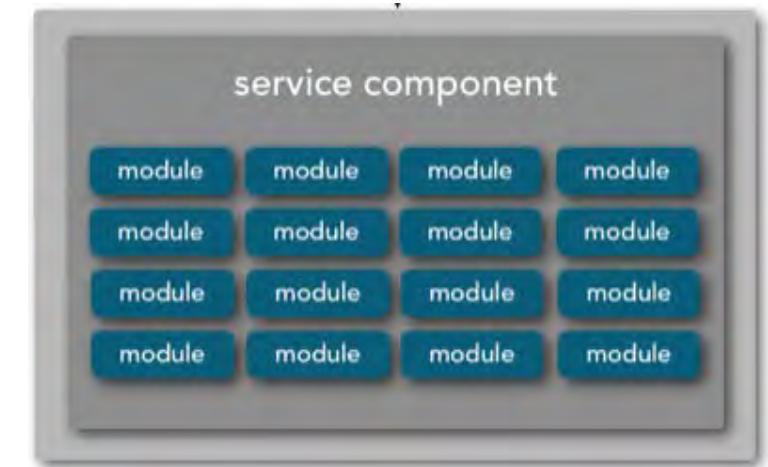
service a



service b

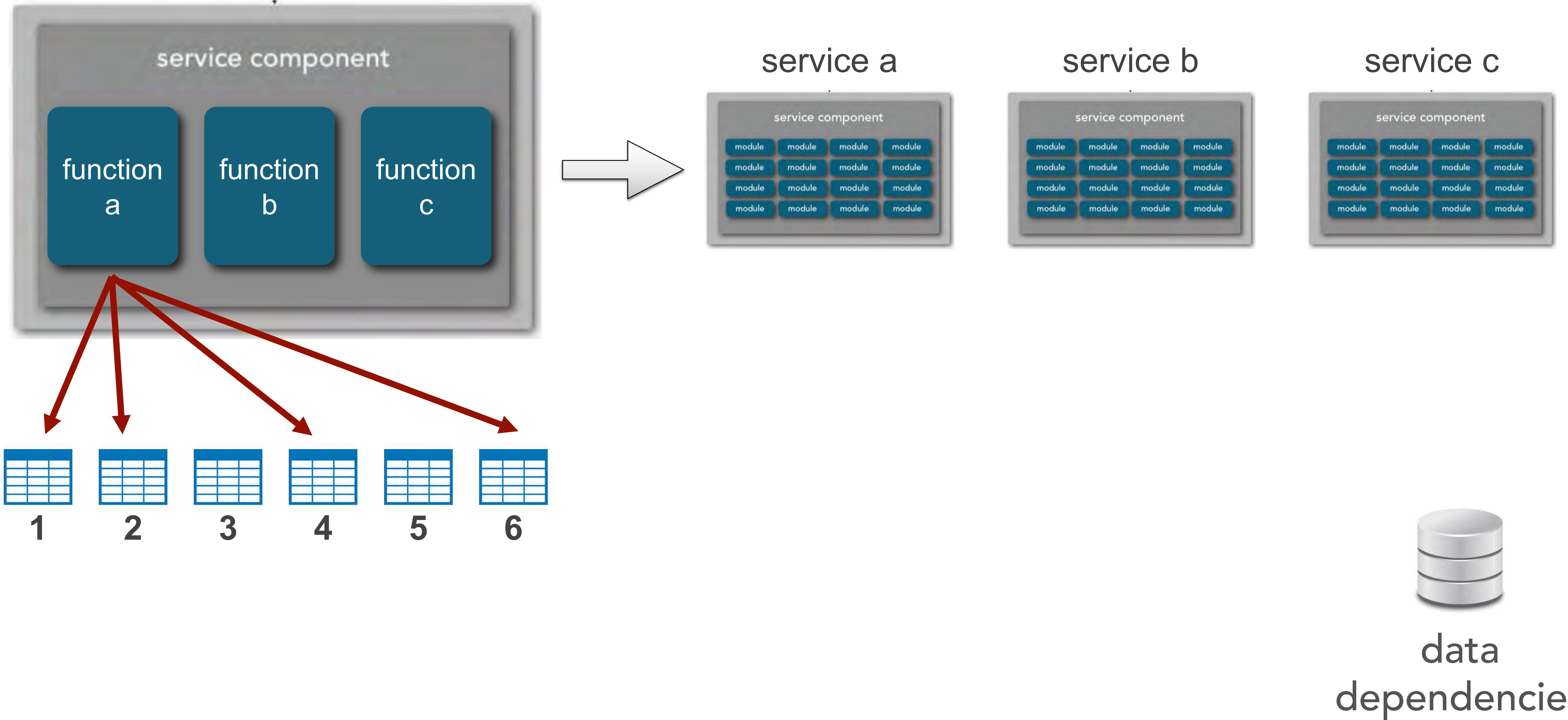


service c

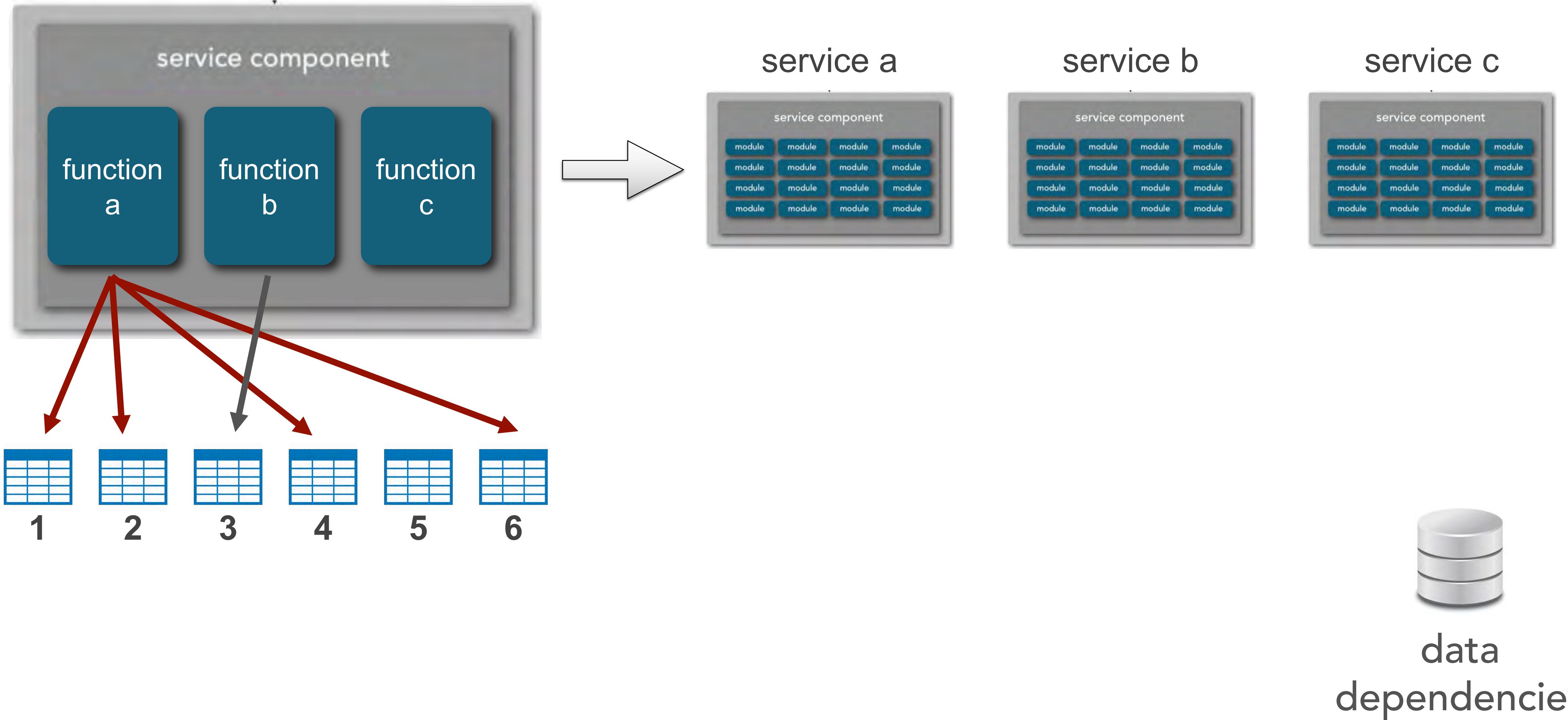


data  
dependencies

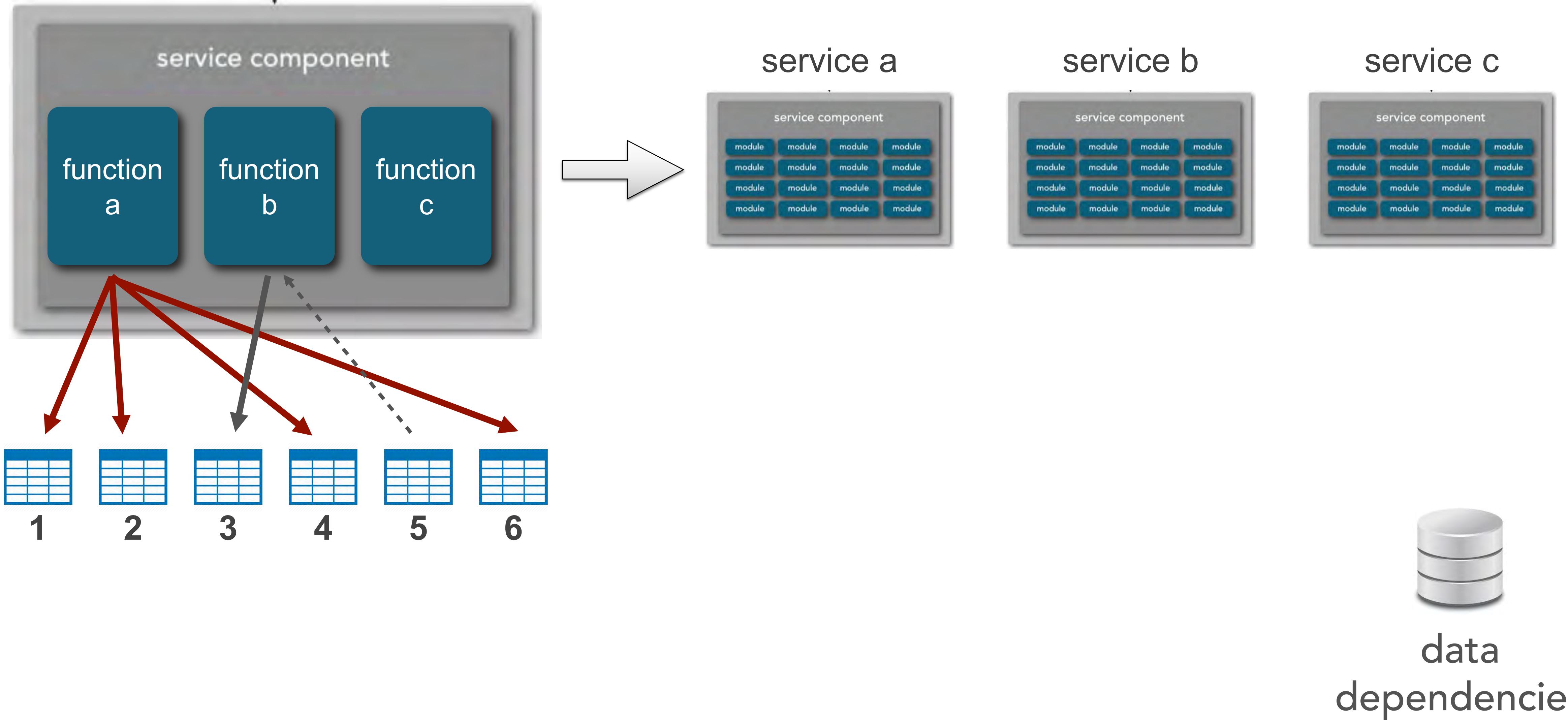
# service granularity



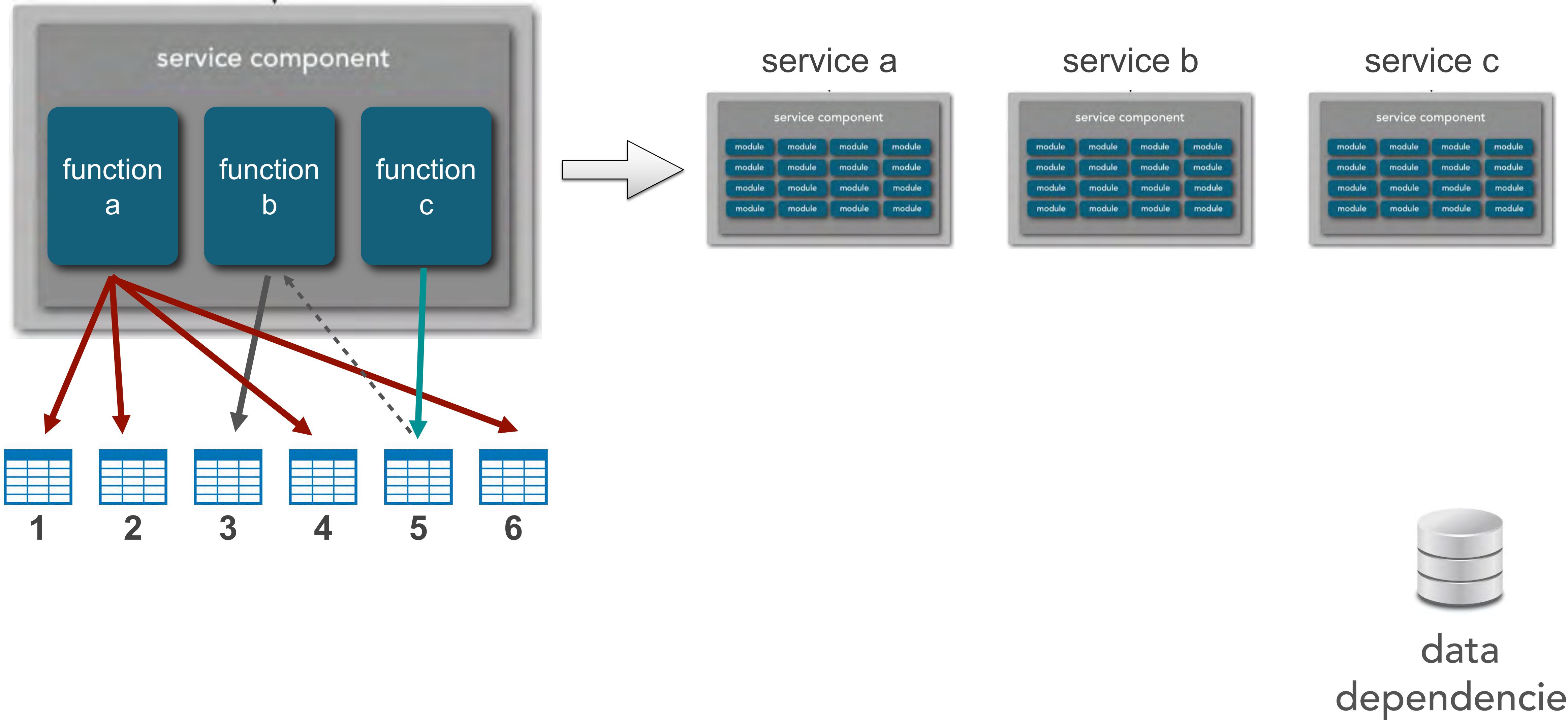
# service granularity



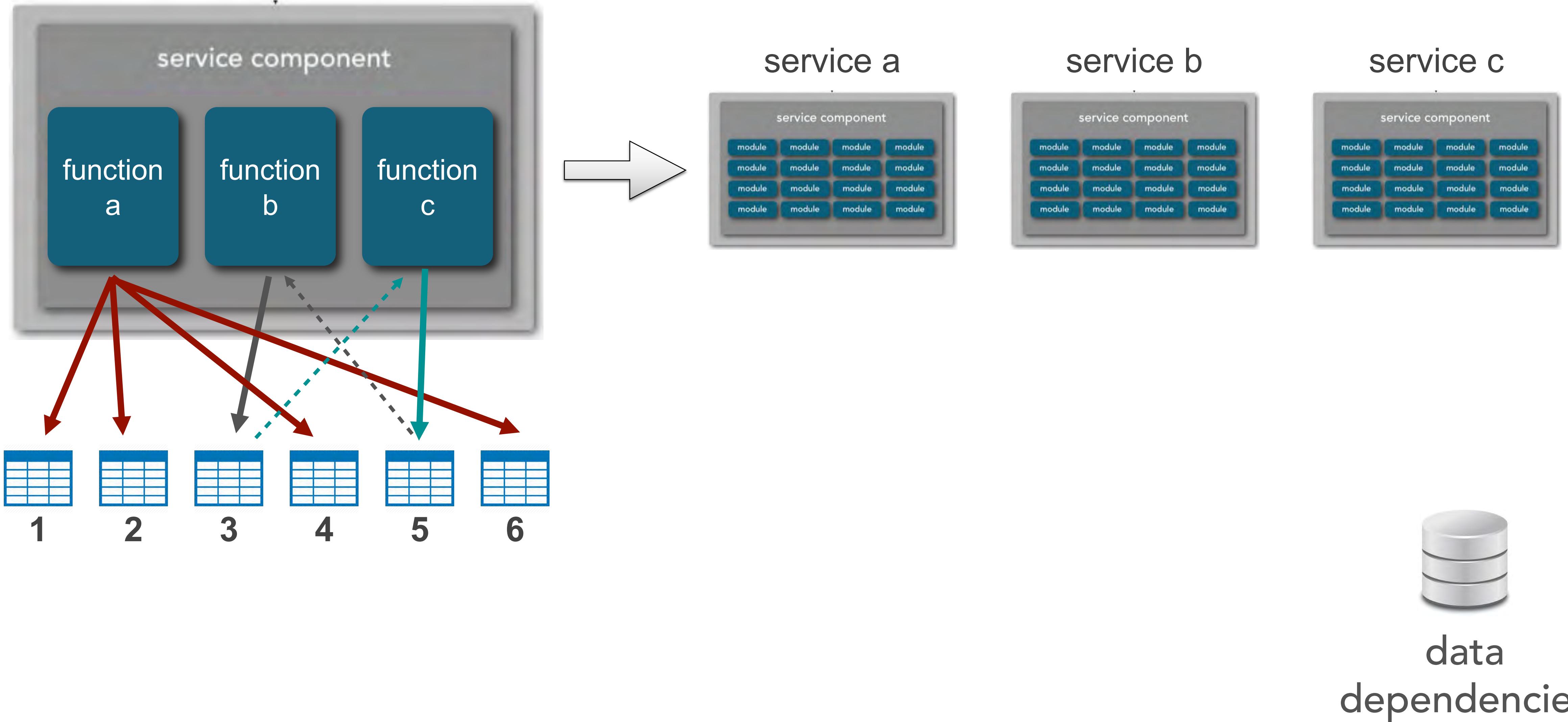
# service granularity



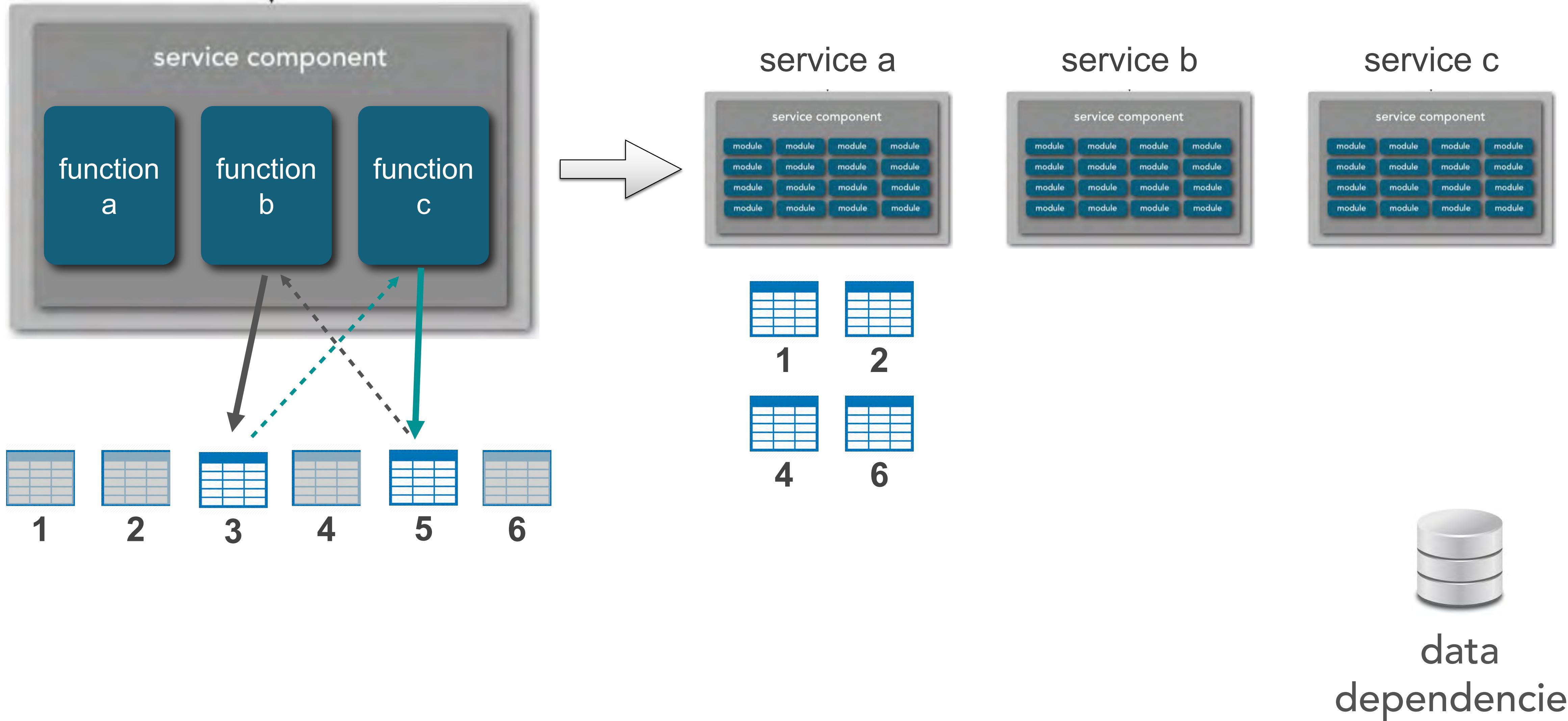
# service granularity



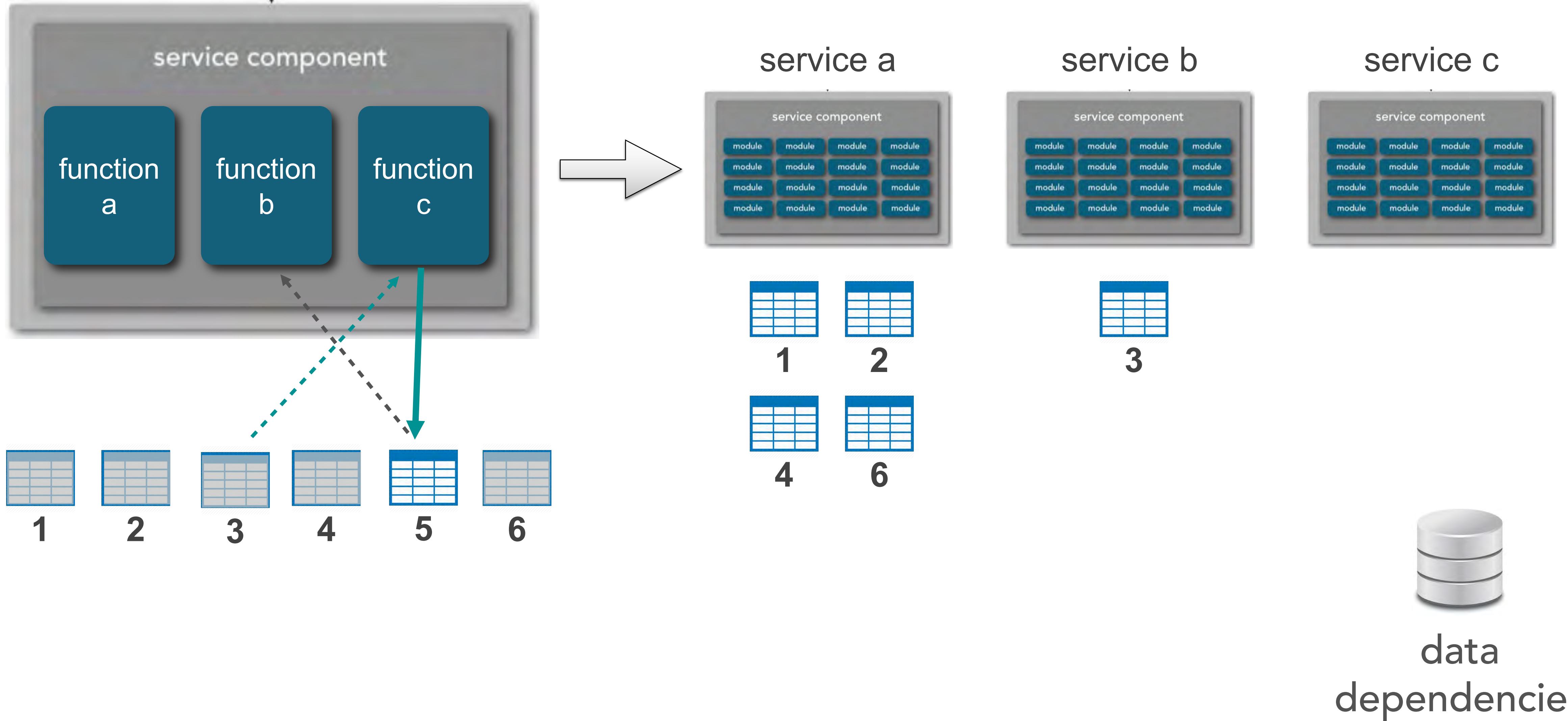
# service granularity



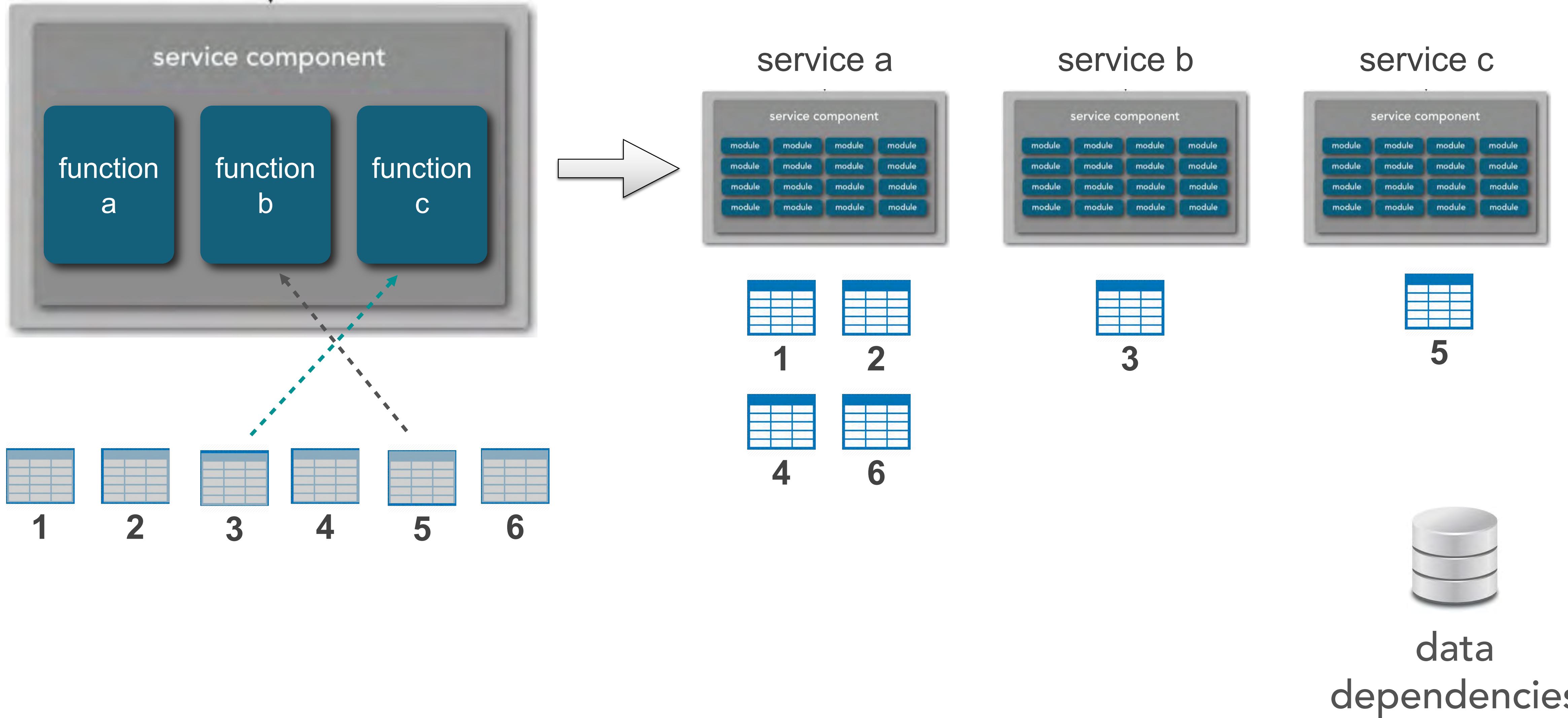
# service granularity



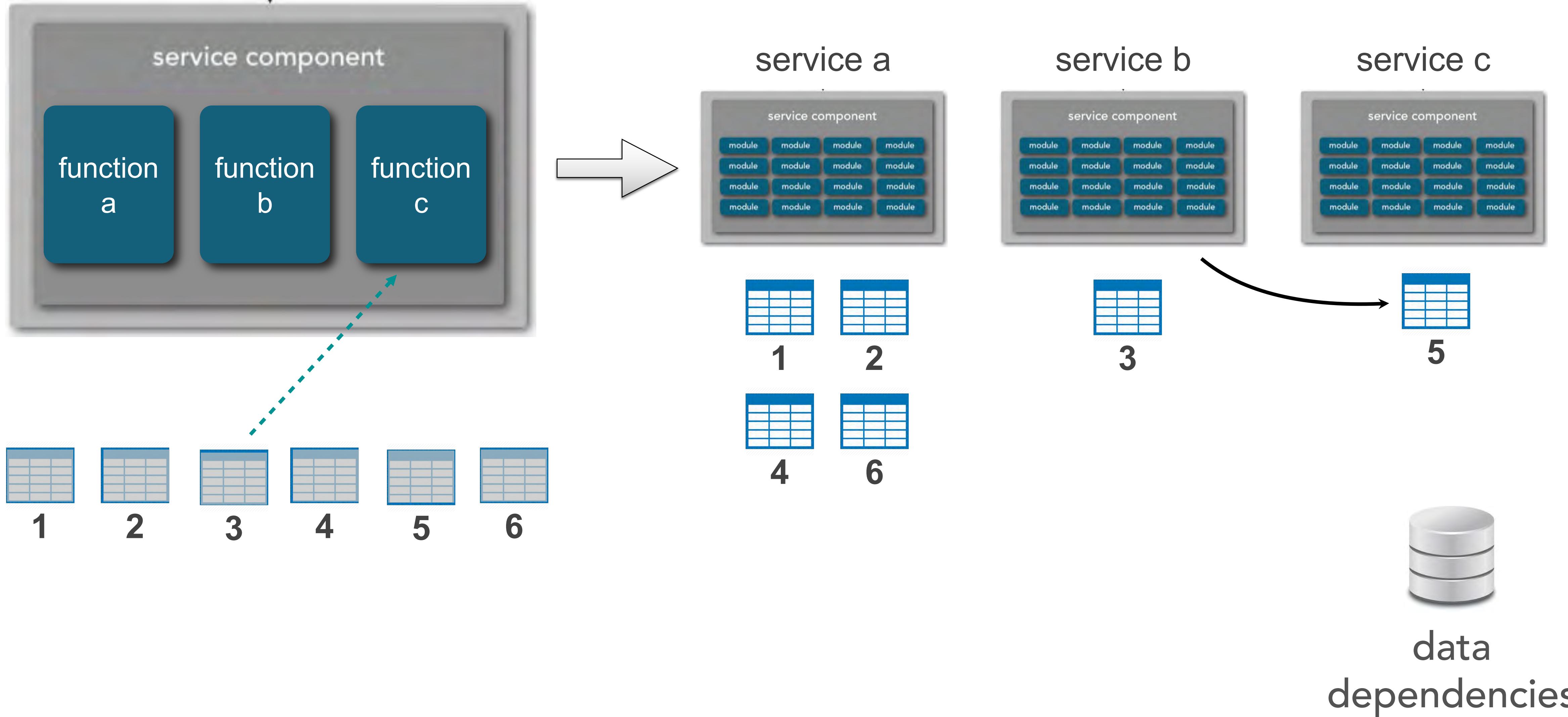
# service granularity



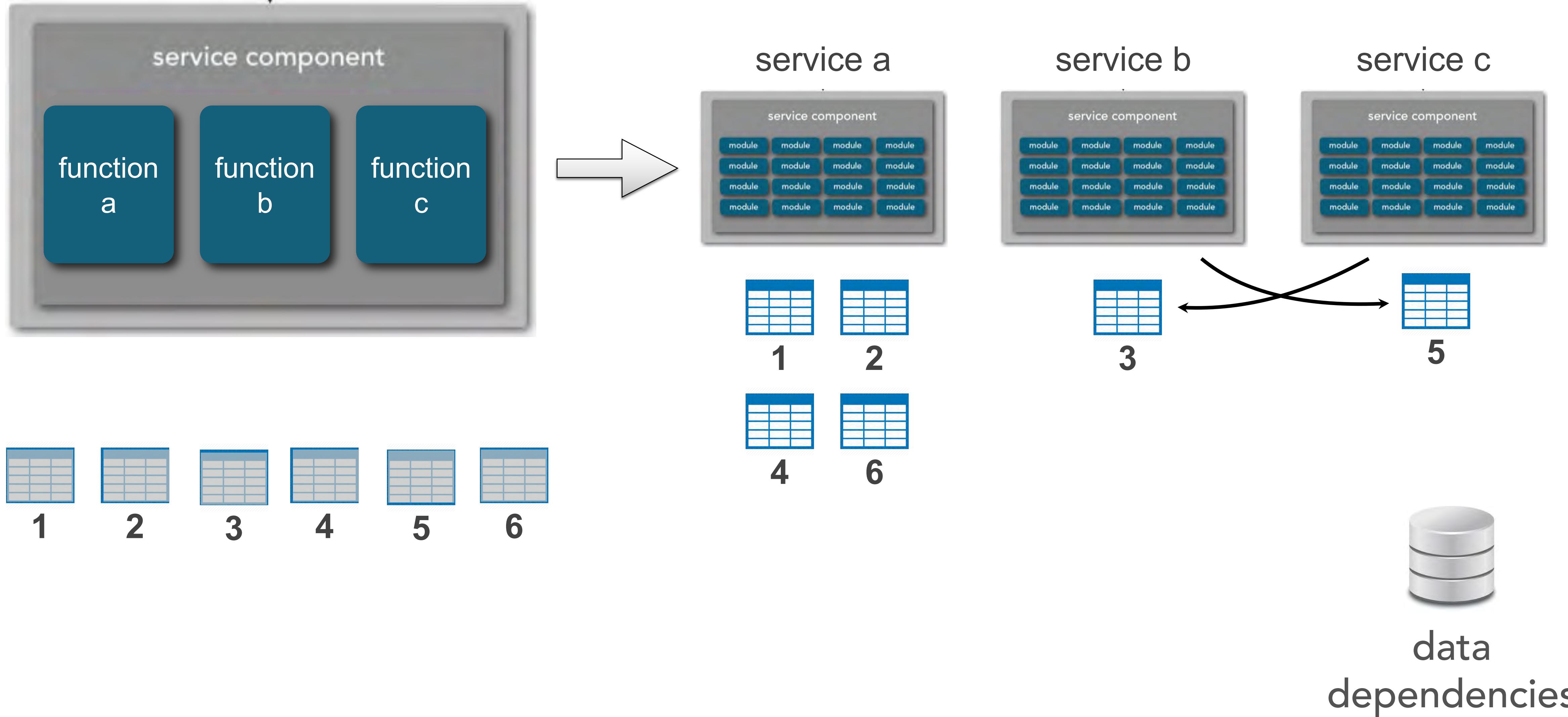
# service granularity



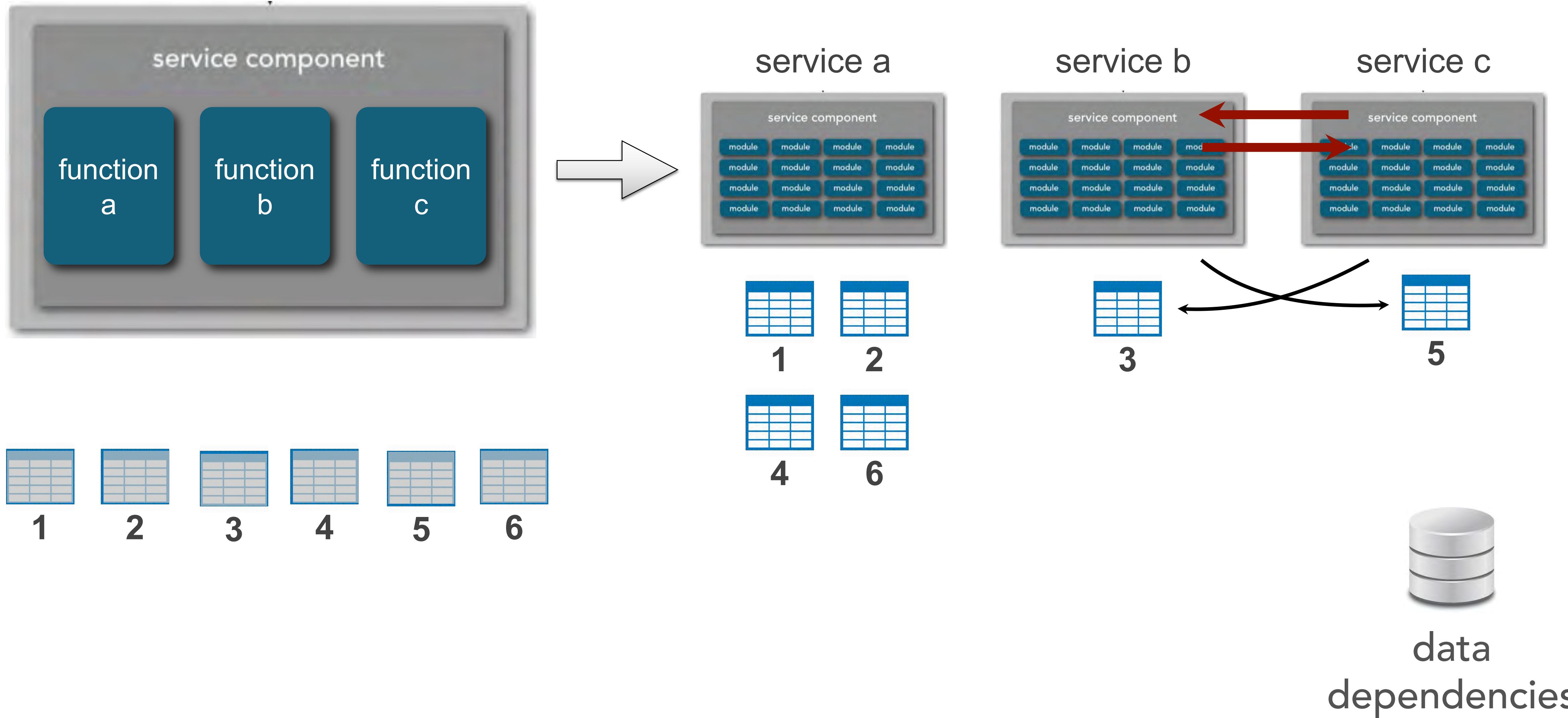
# service granularity



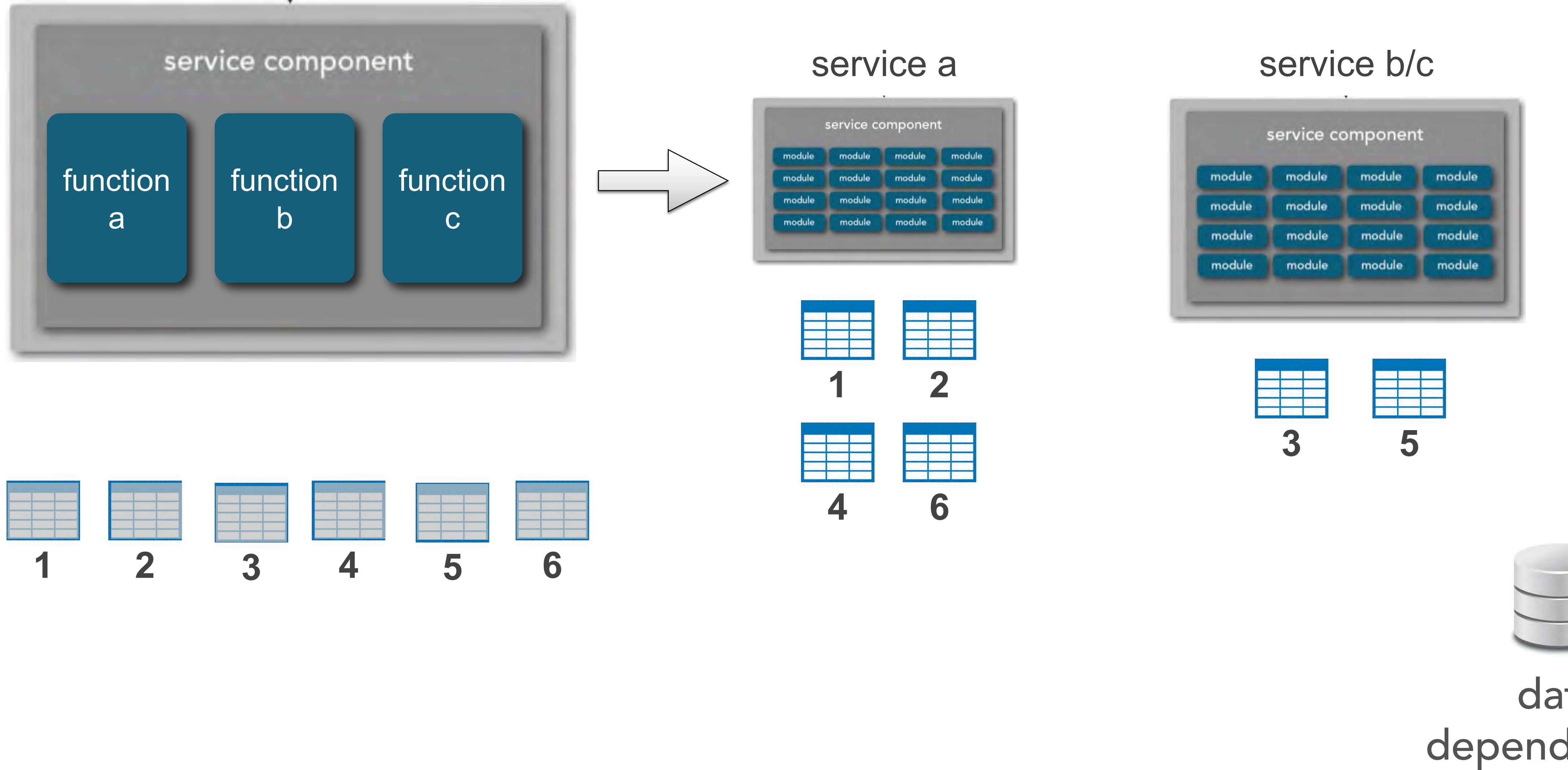
# service granularity



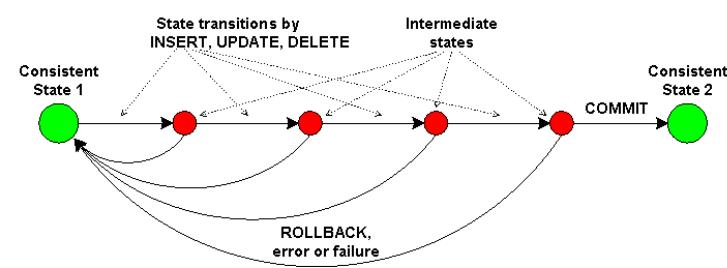
# service granularity



# service granularity



# service granularity



database  
transactions

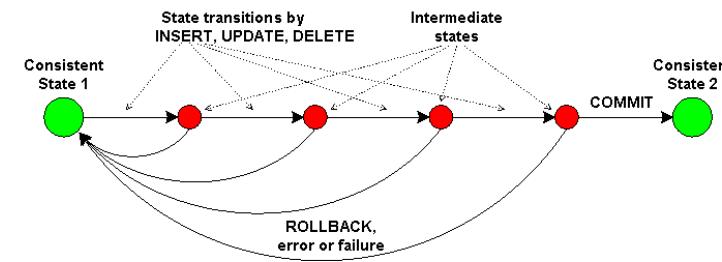
data  
dependencies



granularity integrators

*“when should I consider putting services back together?”*

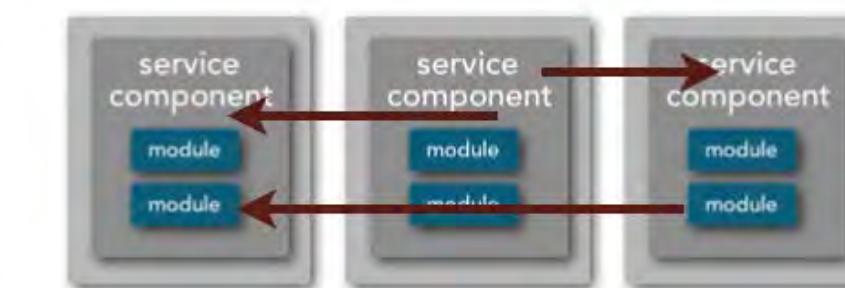
# service granularity



database  
transactions



data  
dependencies



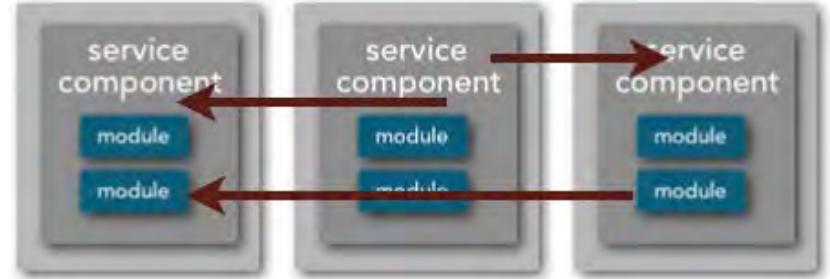
workflow and  
choreography



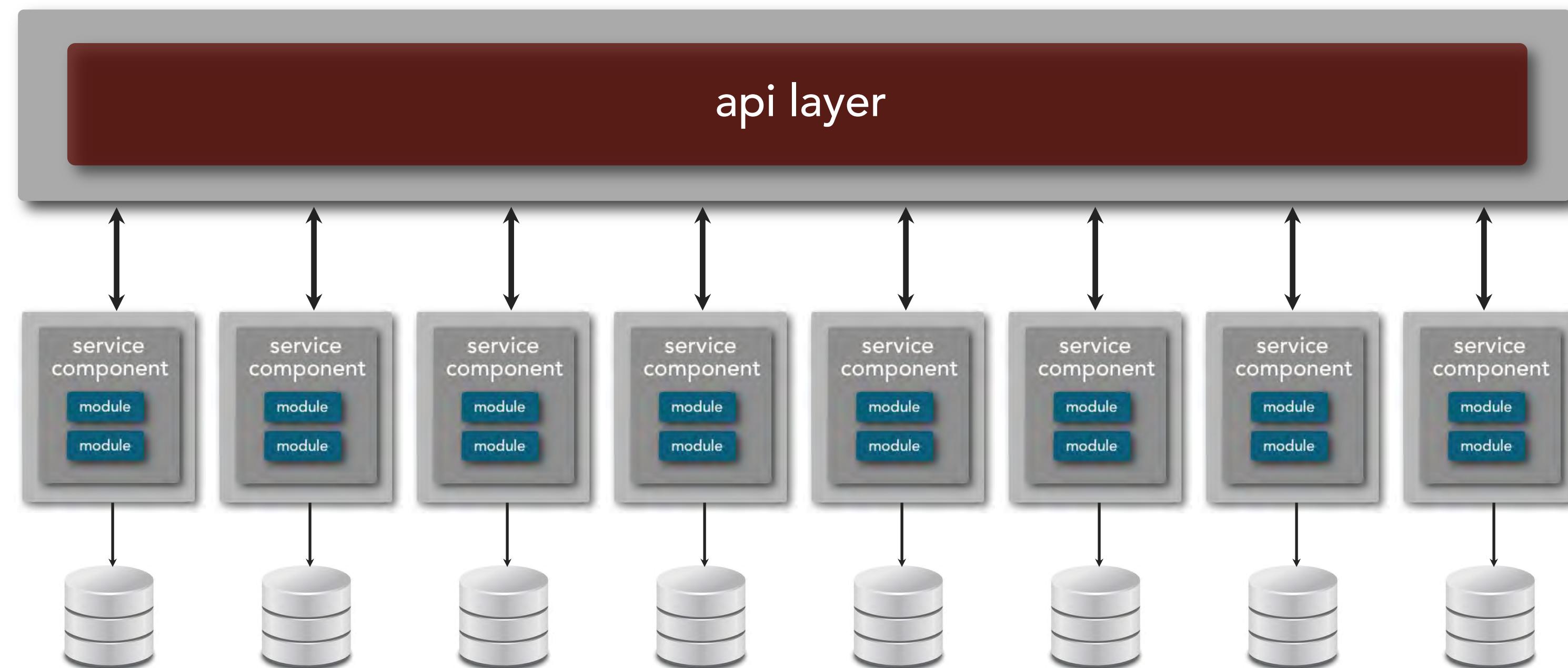
granularity integrators

*"when should I consider putting services back together?"*

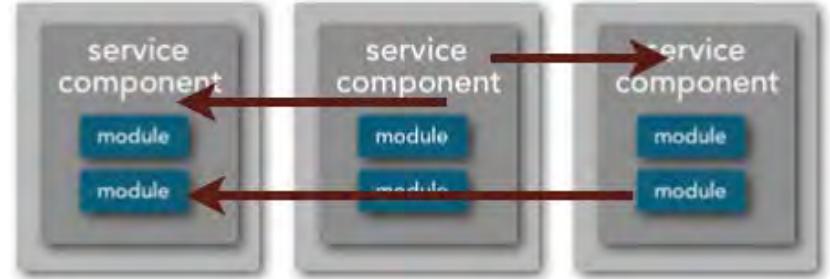
# service granularity



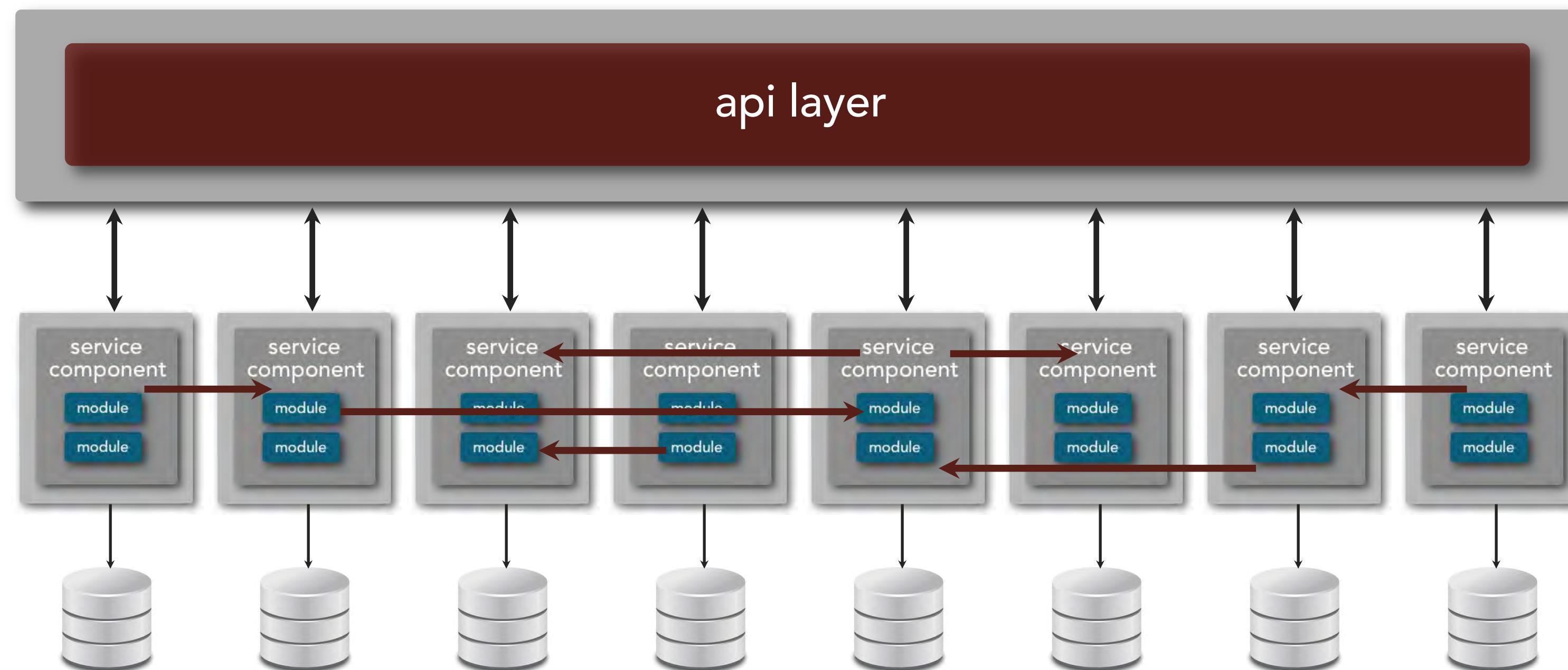
workflow and  
choreography



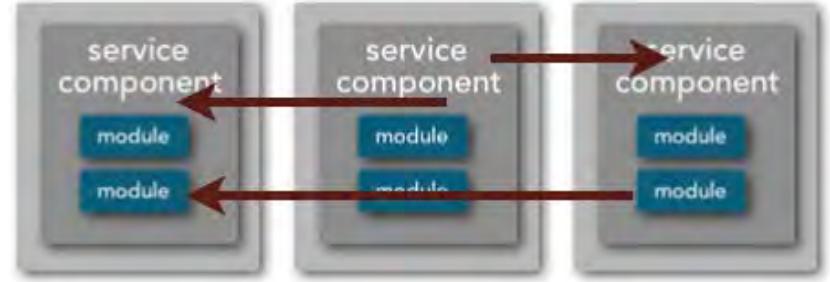
# service granularity



workflow and  
choreography

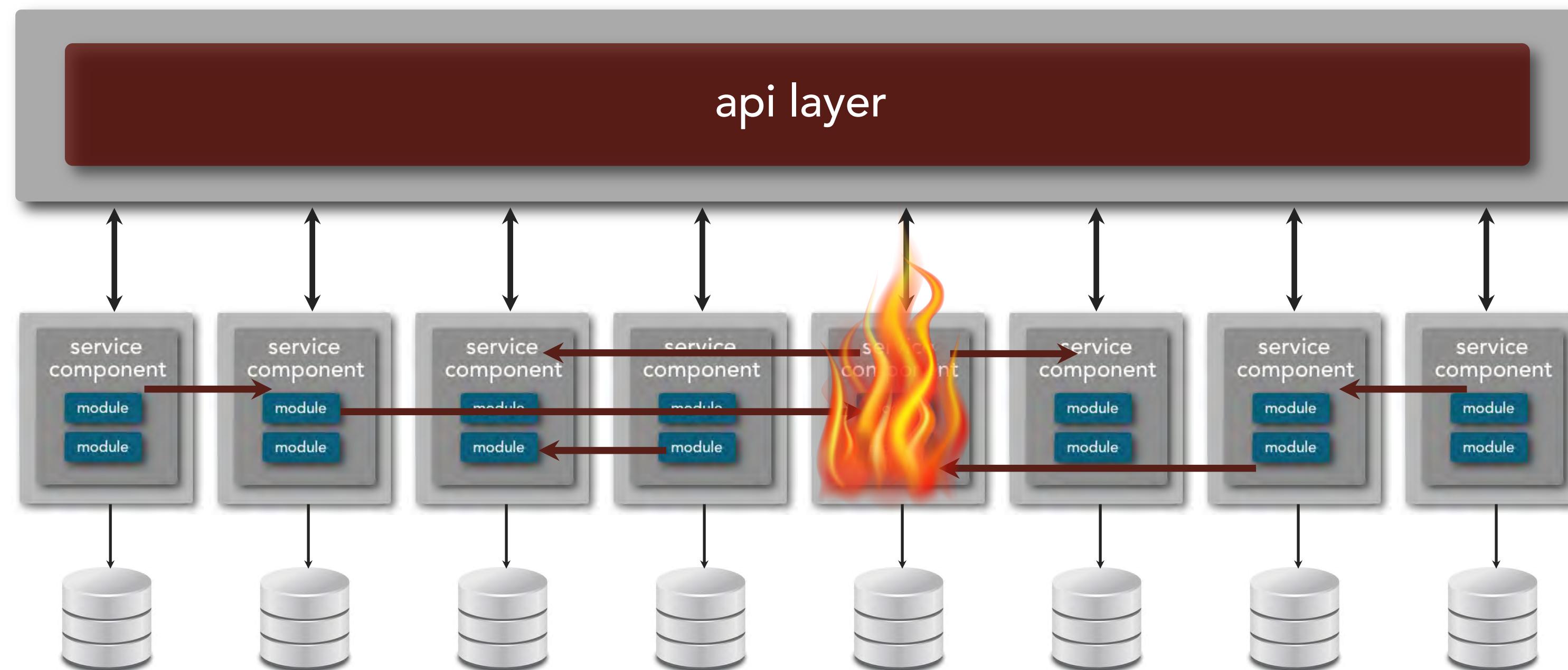


# service granularity

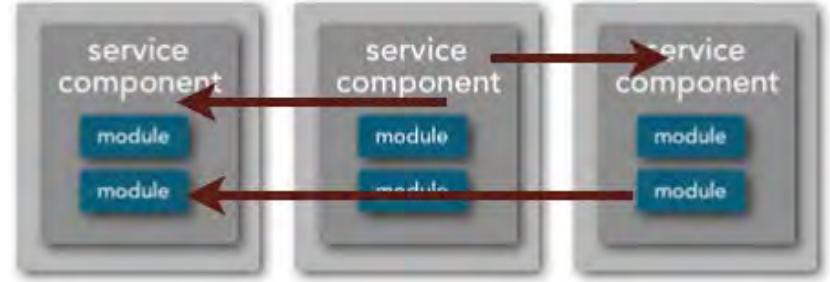


workflow and  
choreography

fault tolerance (availability)

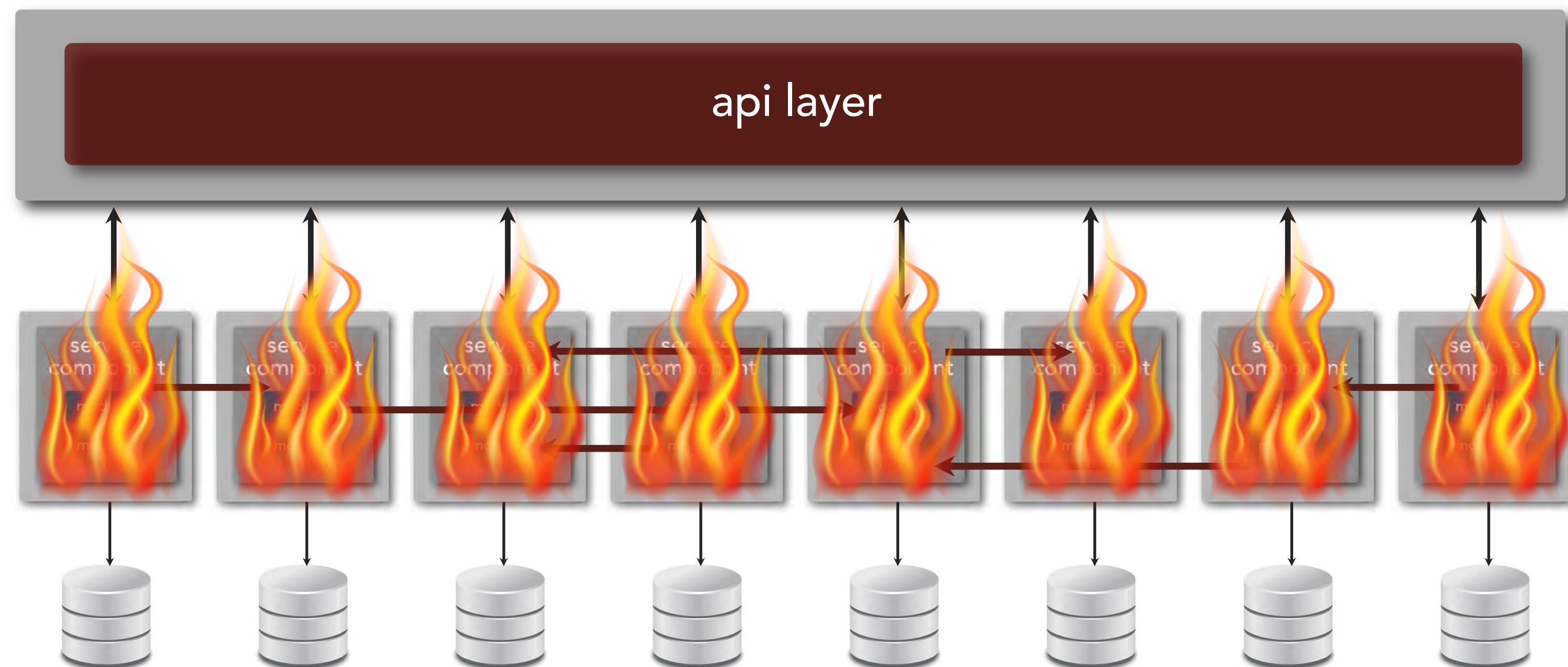


# service granularity

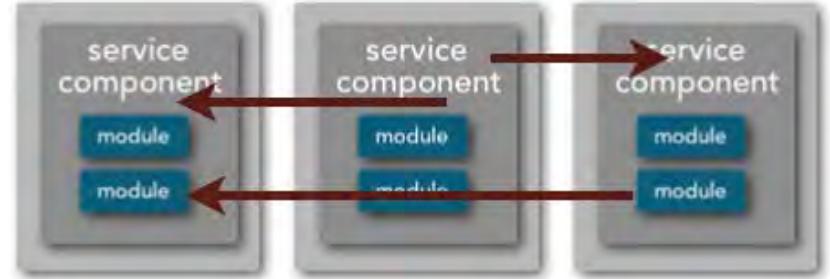


workflow and  
choreography

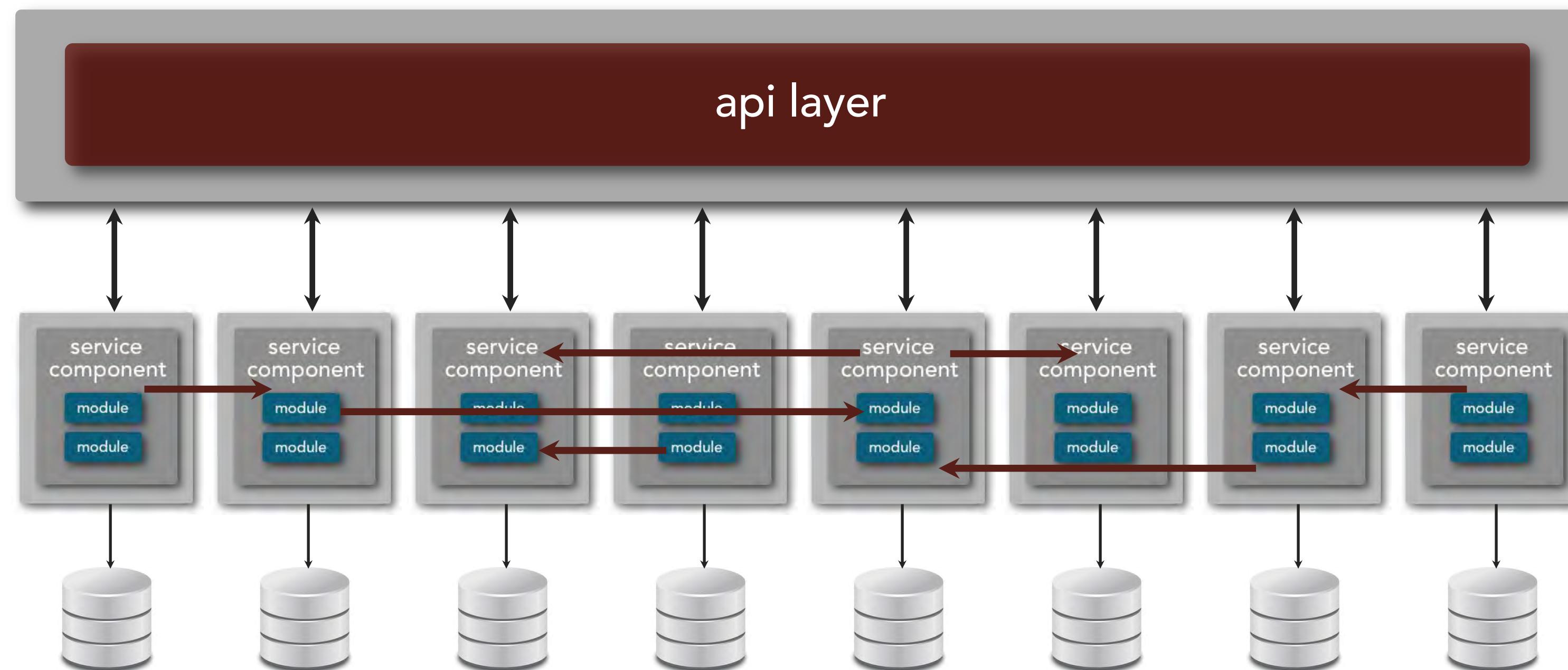
fault tolerance (availability)



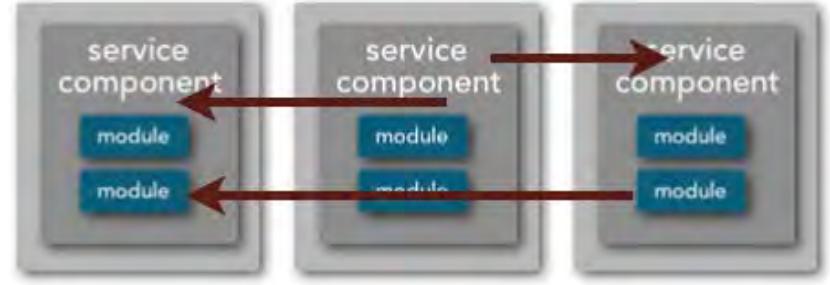
# service granularity



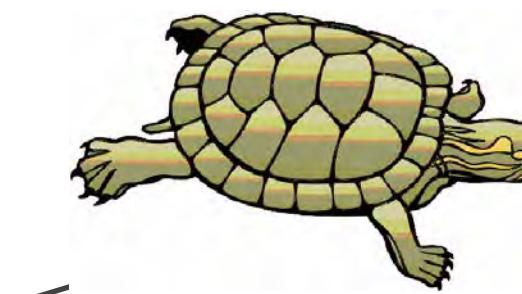
workflow and  
choreography



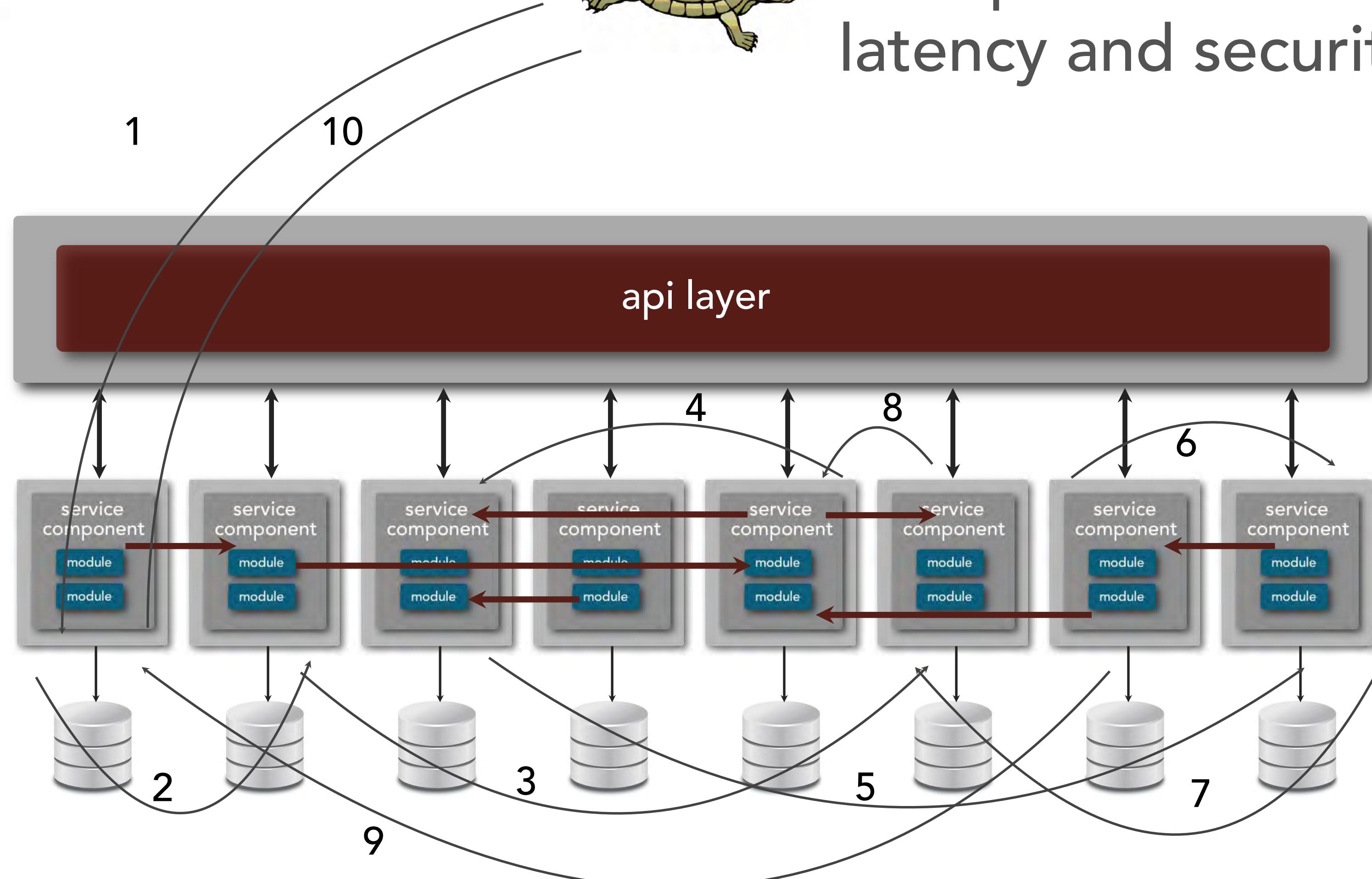
# service granularity



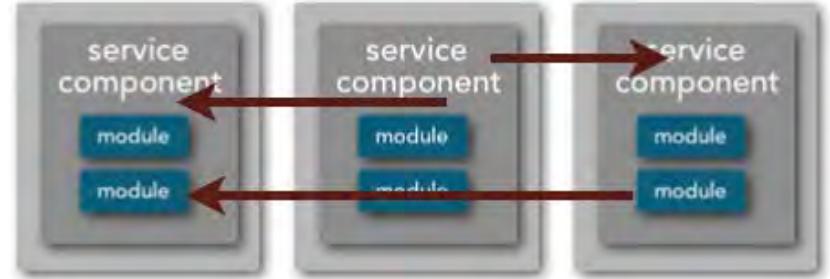
workflow and  
choreography



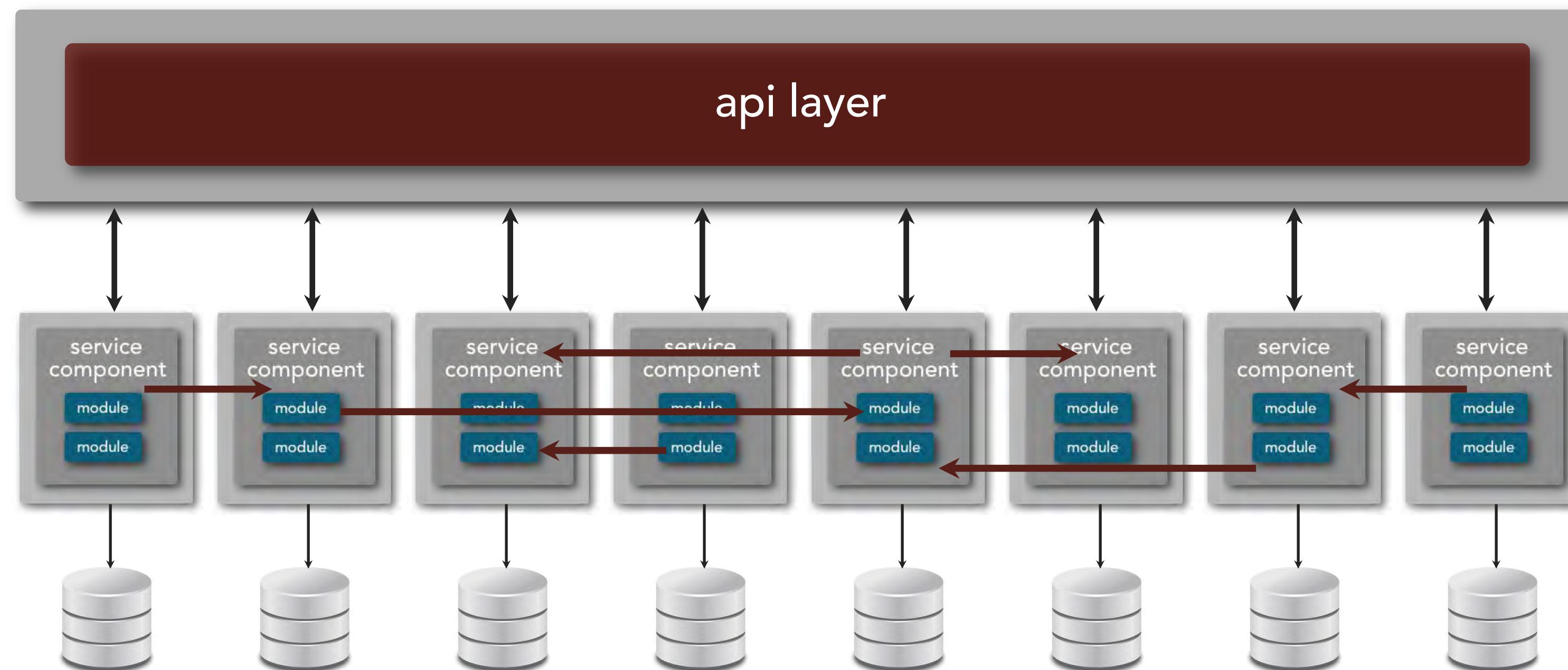
slow performance due to  
latency and security



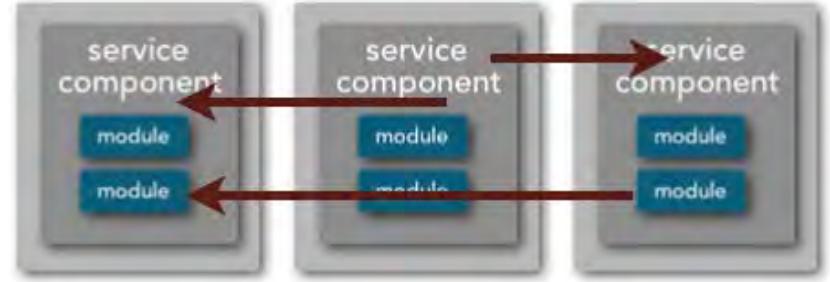
# service granularity



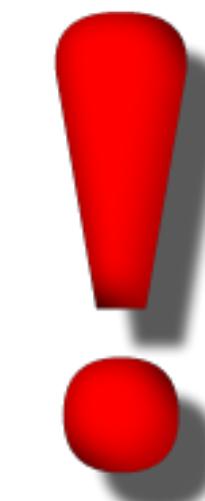
workflow and  
choreography



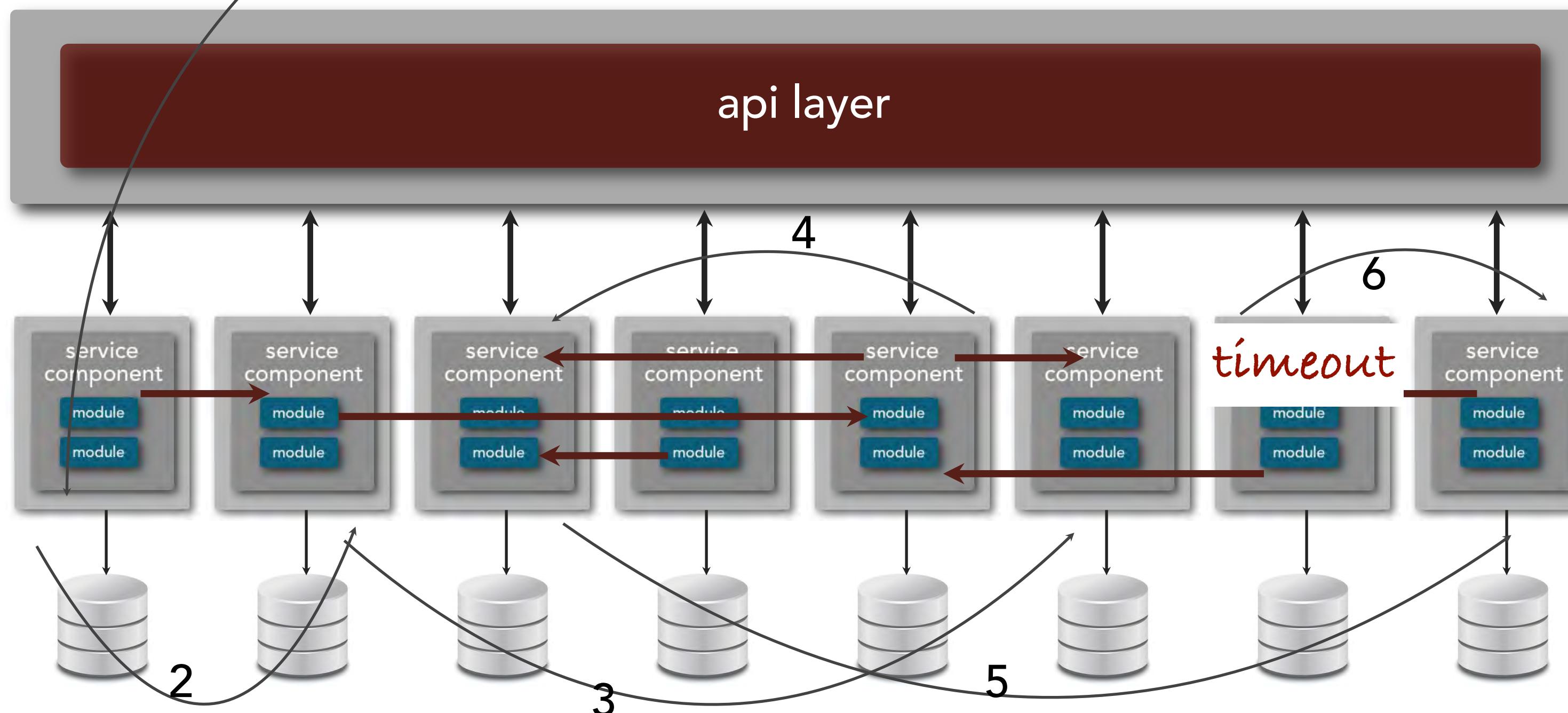
# service granularity



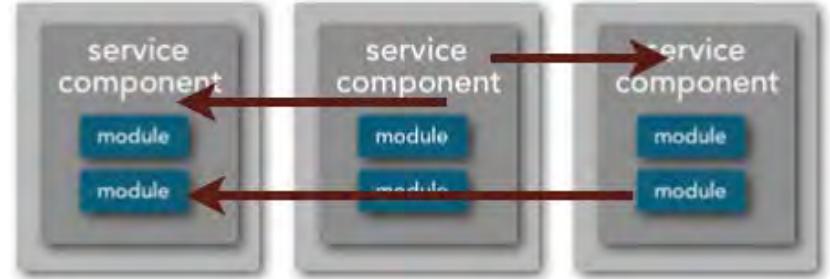
workflow and  
choreography



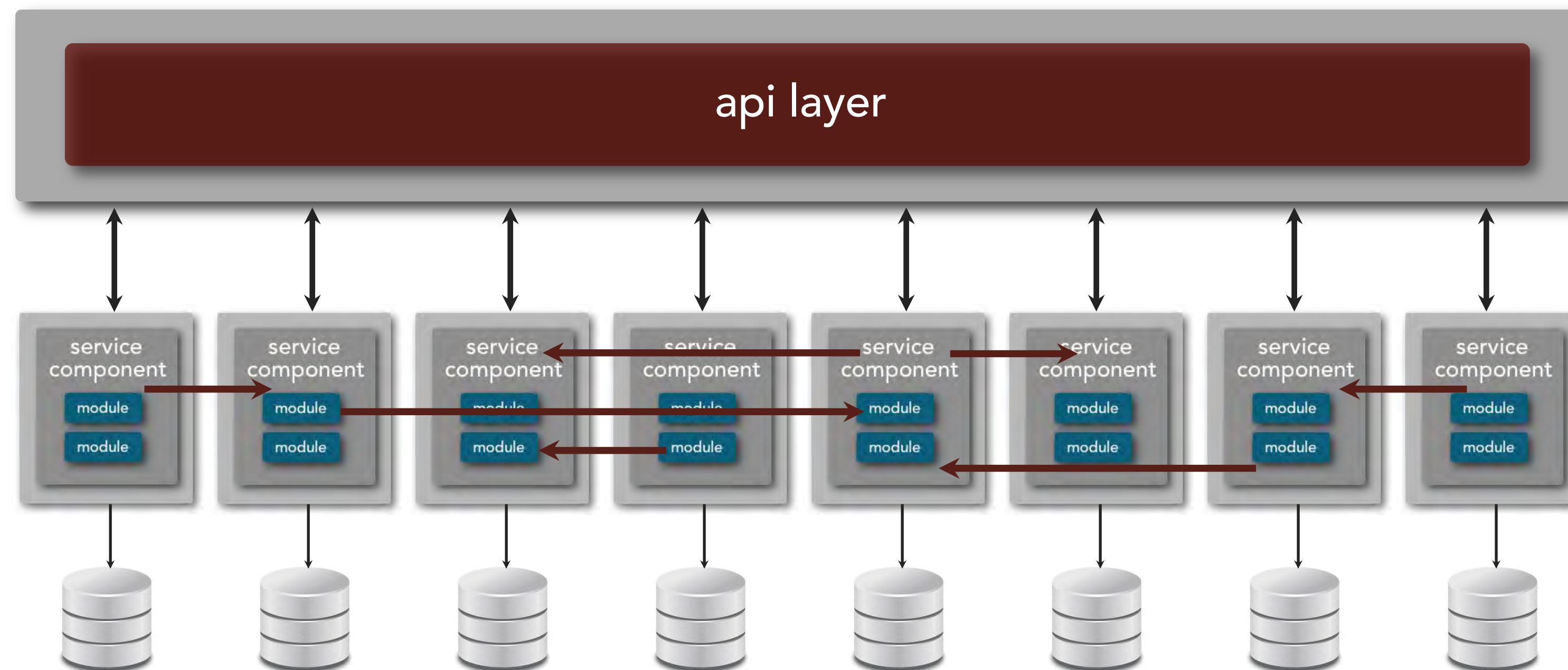
reliability and data consistency



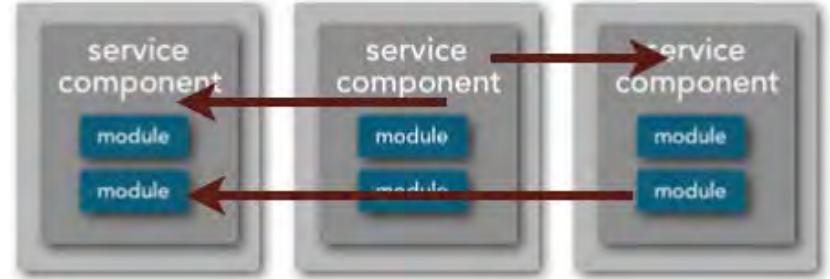
# service granularity



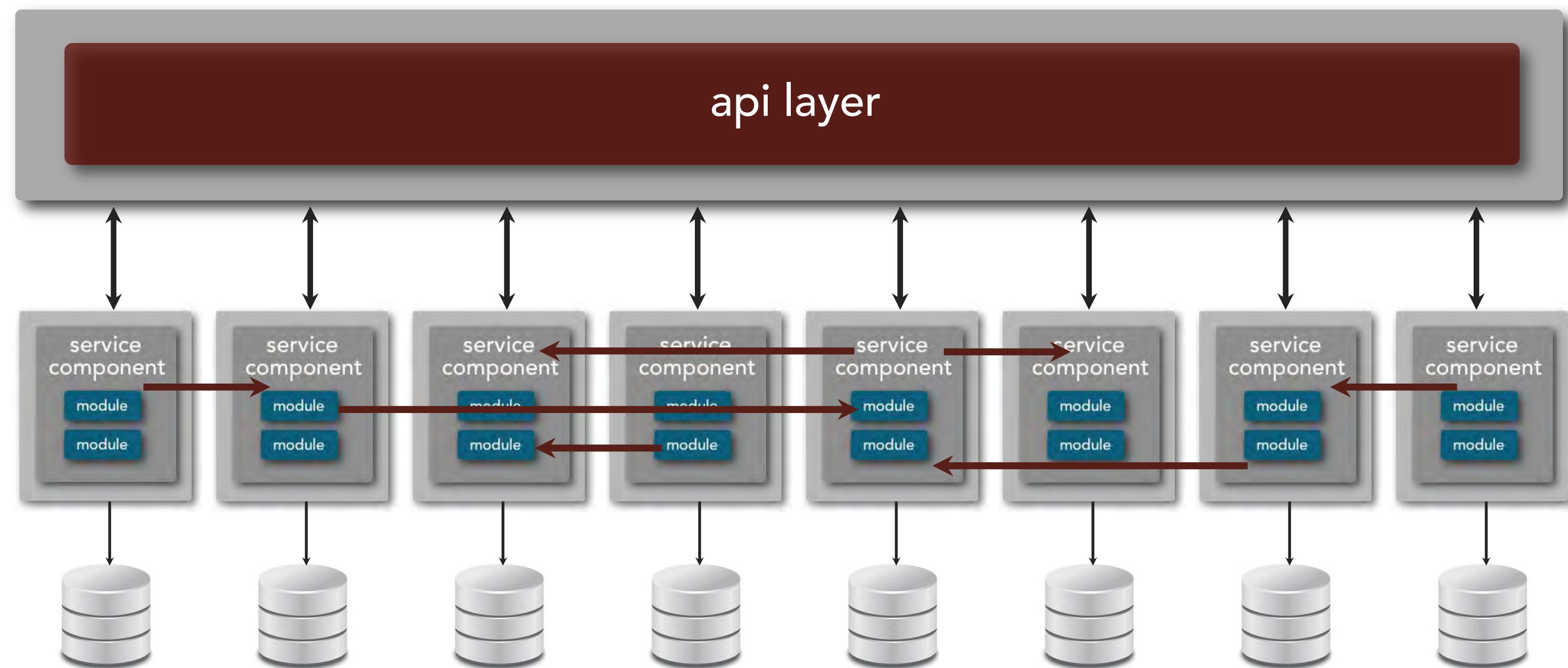
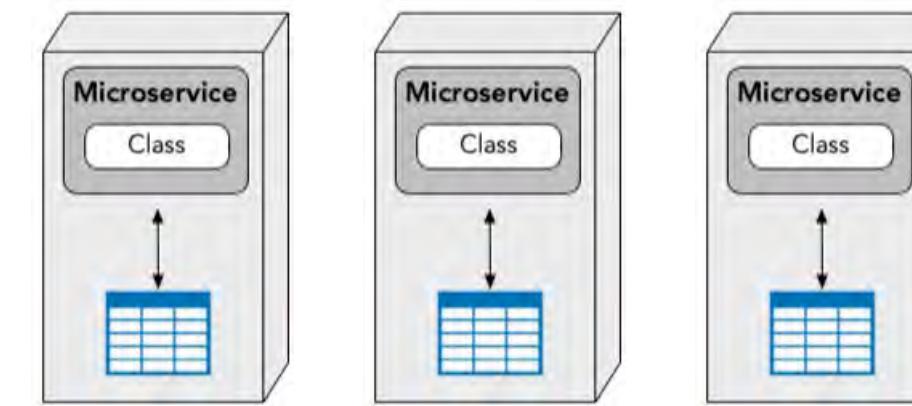
workflow and  
choreography



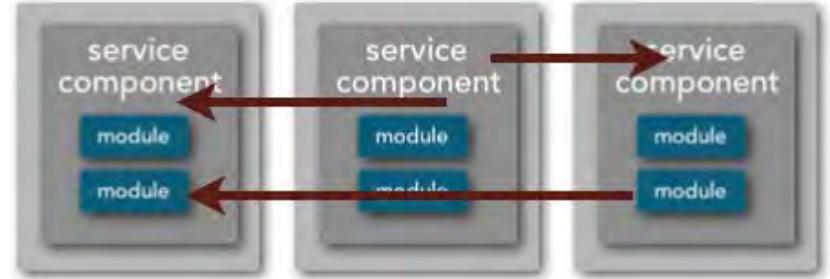
# service granularity



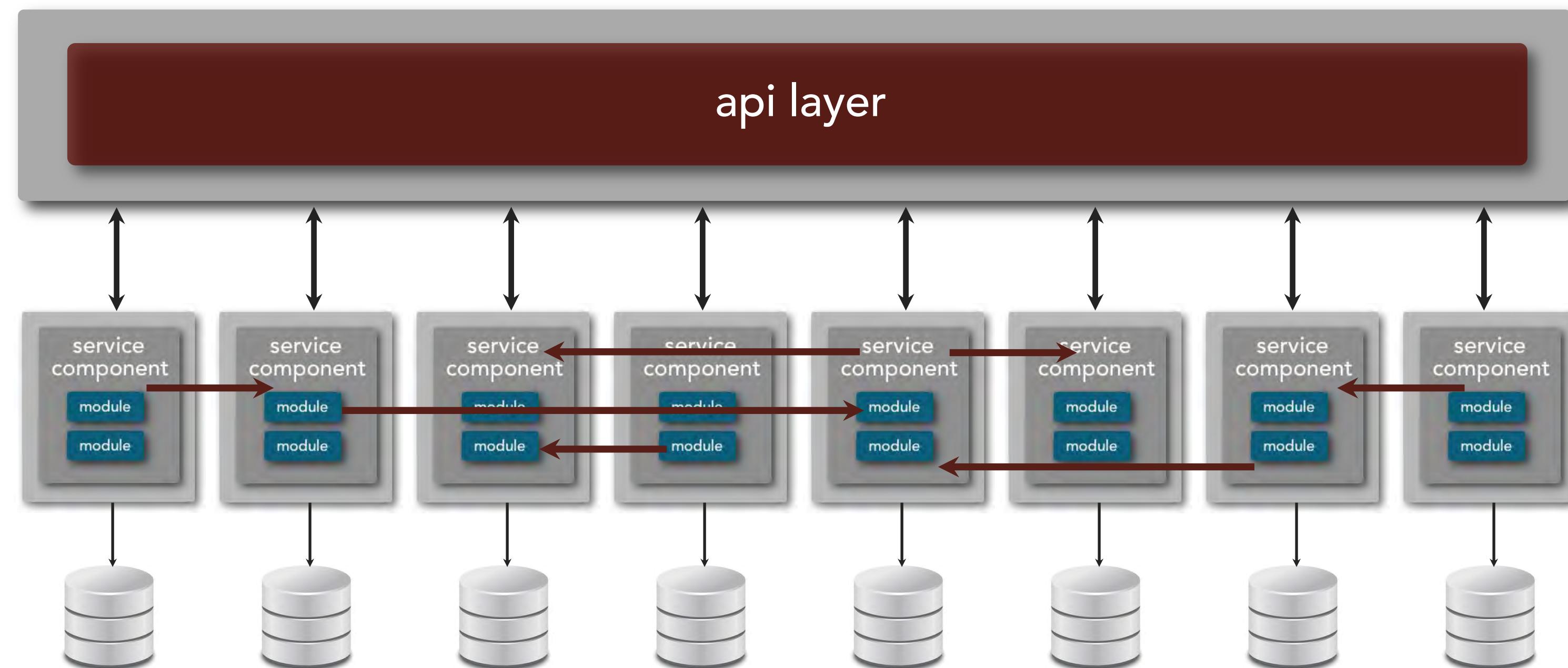
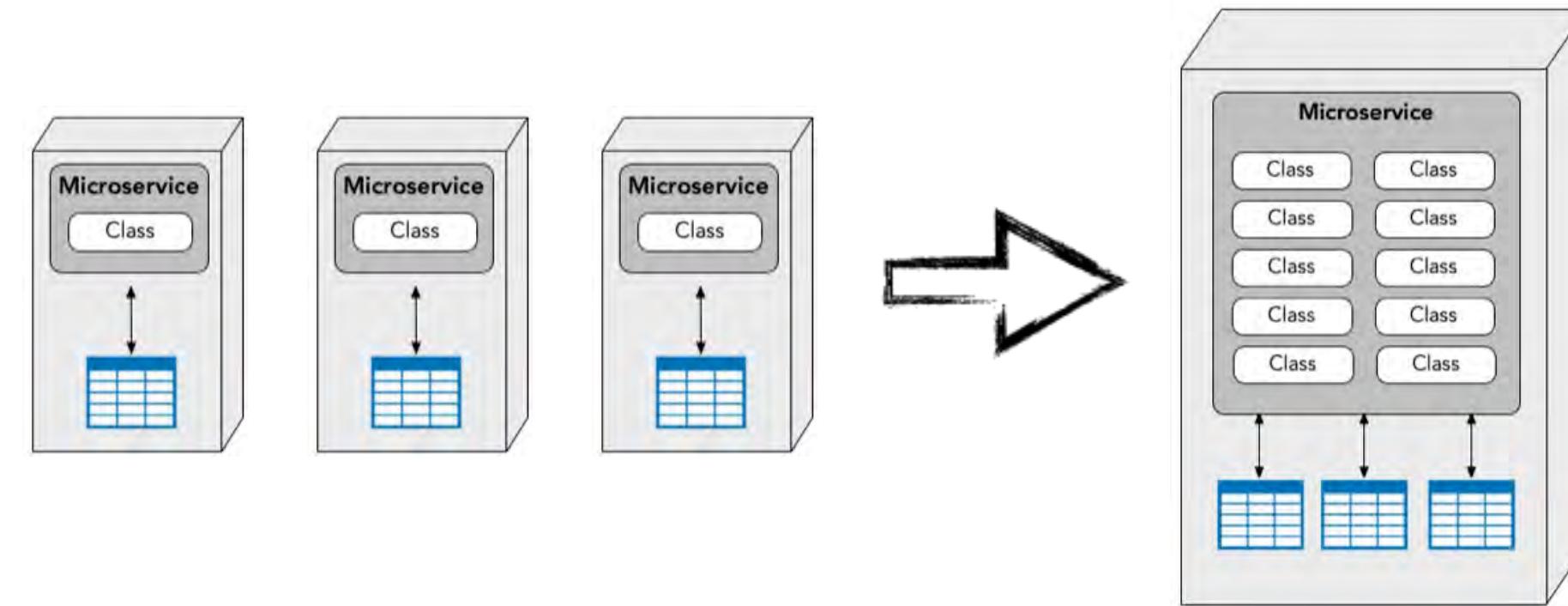
workflow and  
choreography



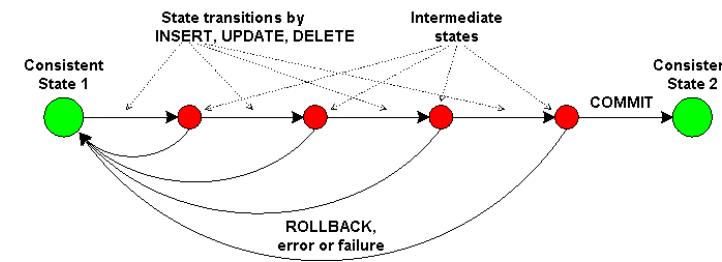
# service granularity



workflow and  
choreography



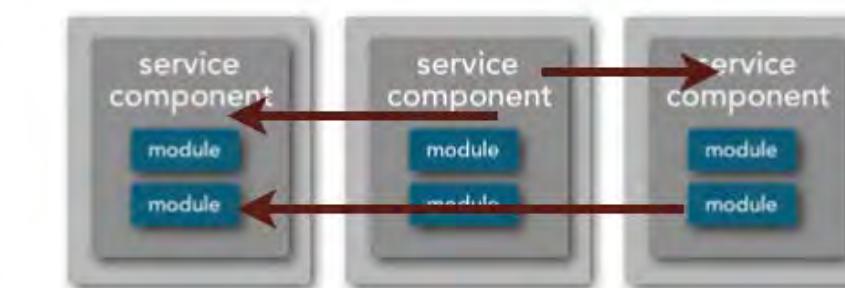
# service granularity



database  
transactions



data  
dependencies



workflow and  
choreography



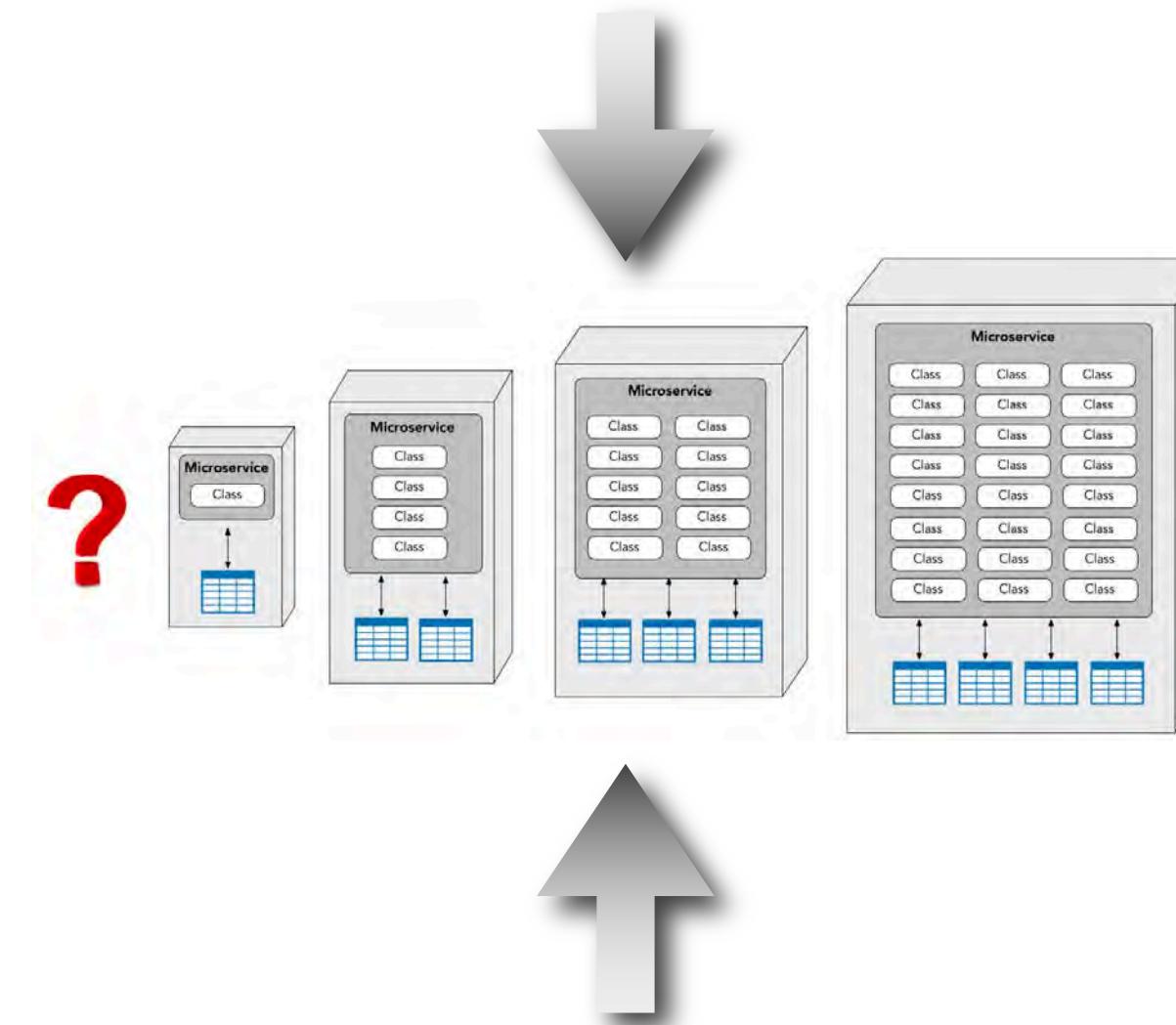
granularity integrators

*"when should I consider putting services back together?"*

# service granularity

## granularity disintegrators

*“when should I consider breaking apart a service?”*

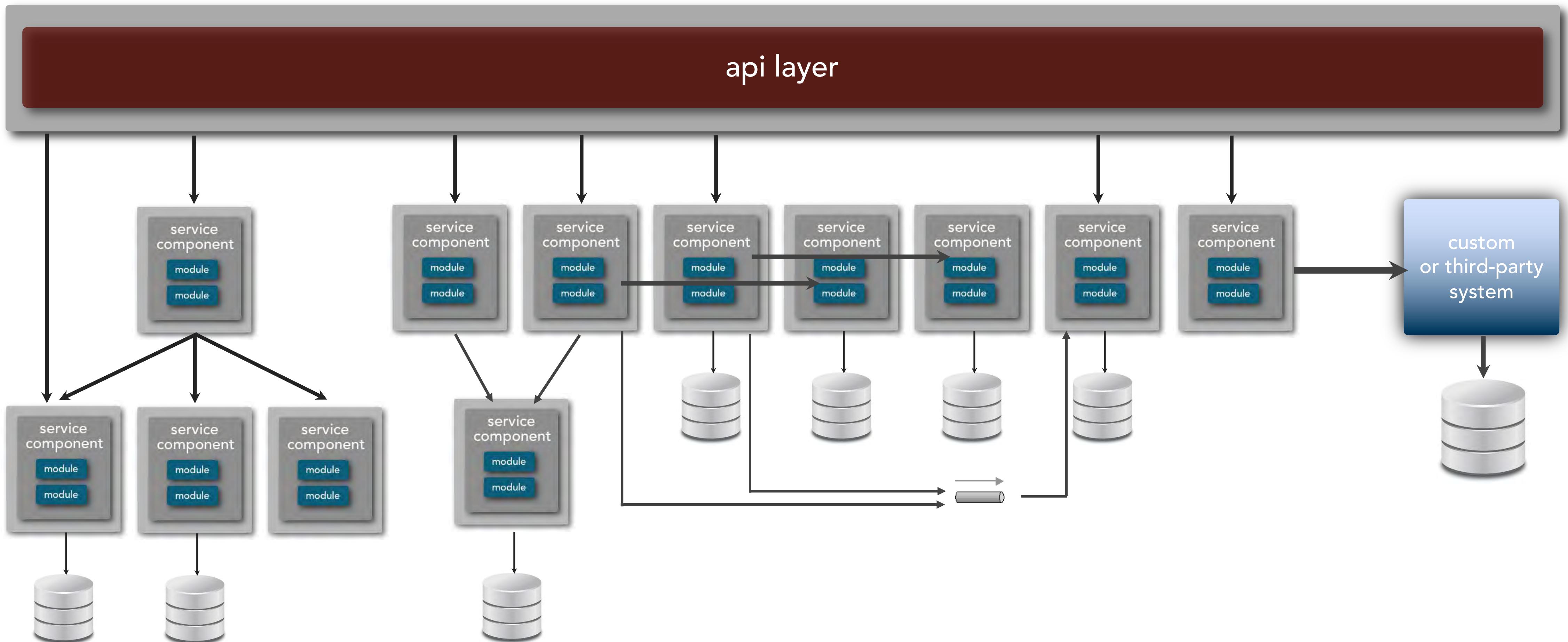


## granularity integrators

*“when should I consider putting services back together?”*

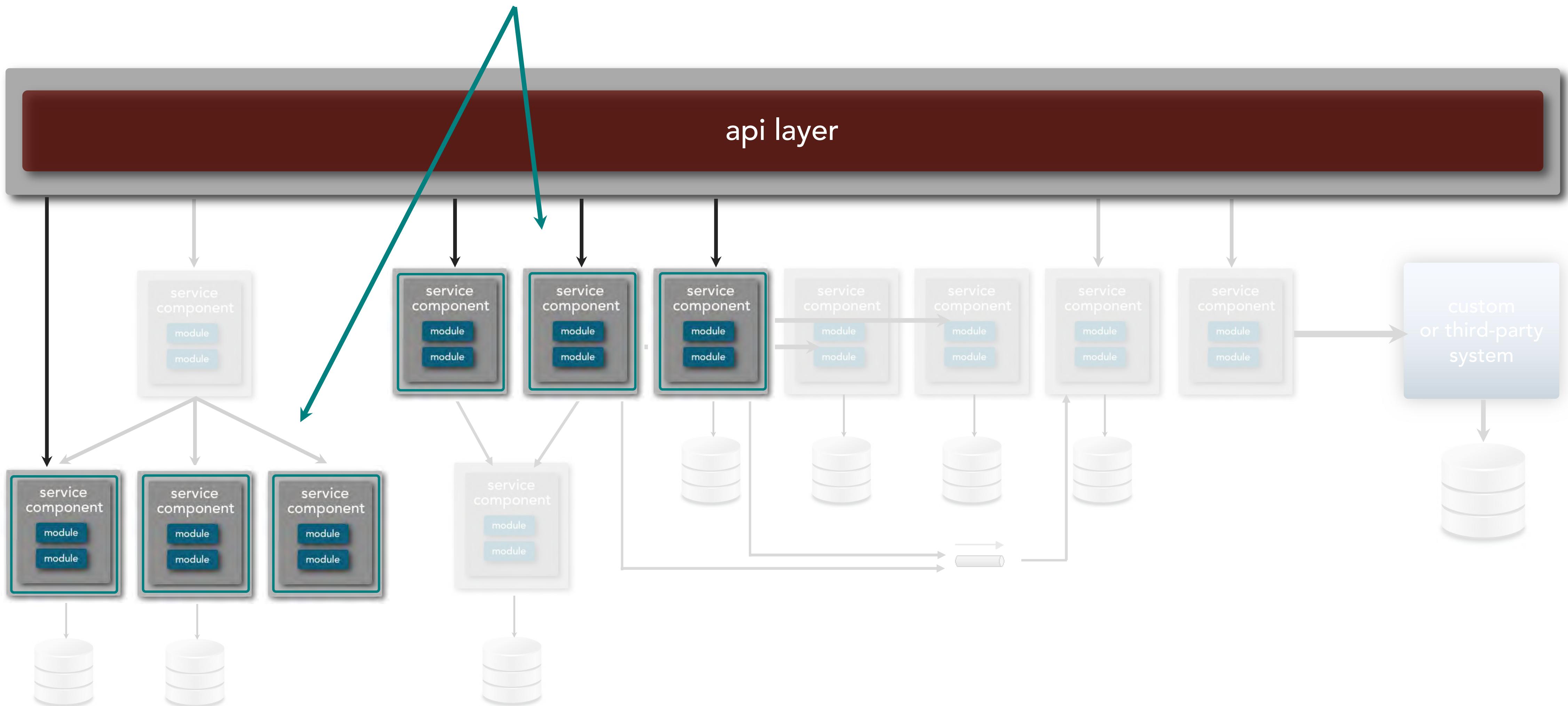
# Service Types

# service types



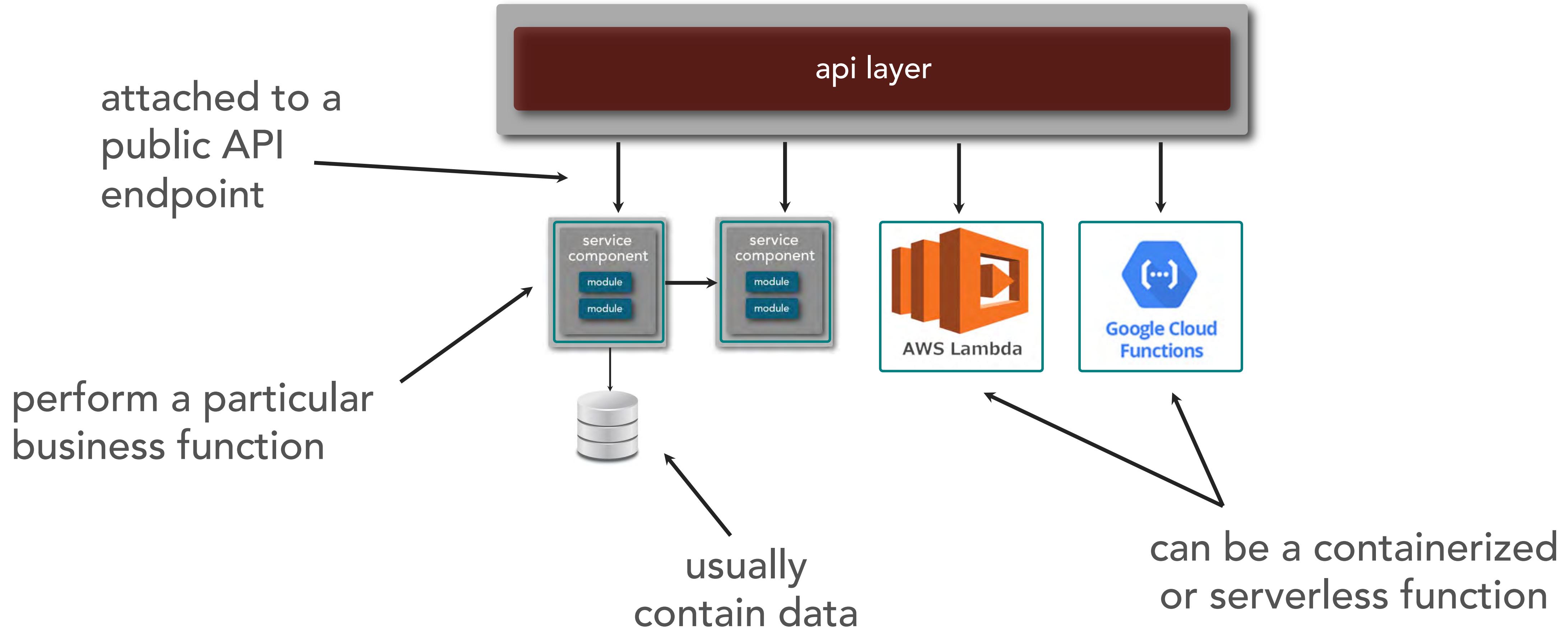
# service types

## functional services



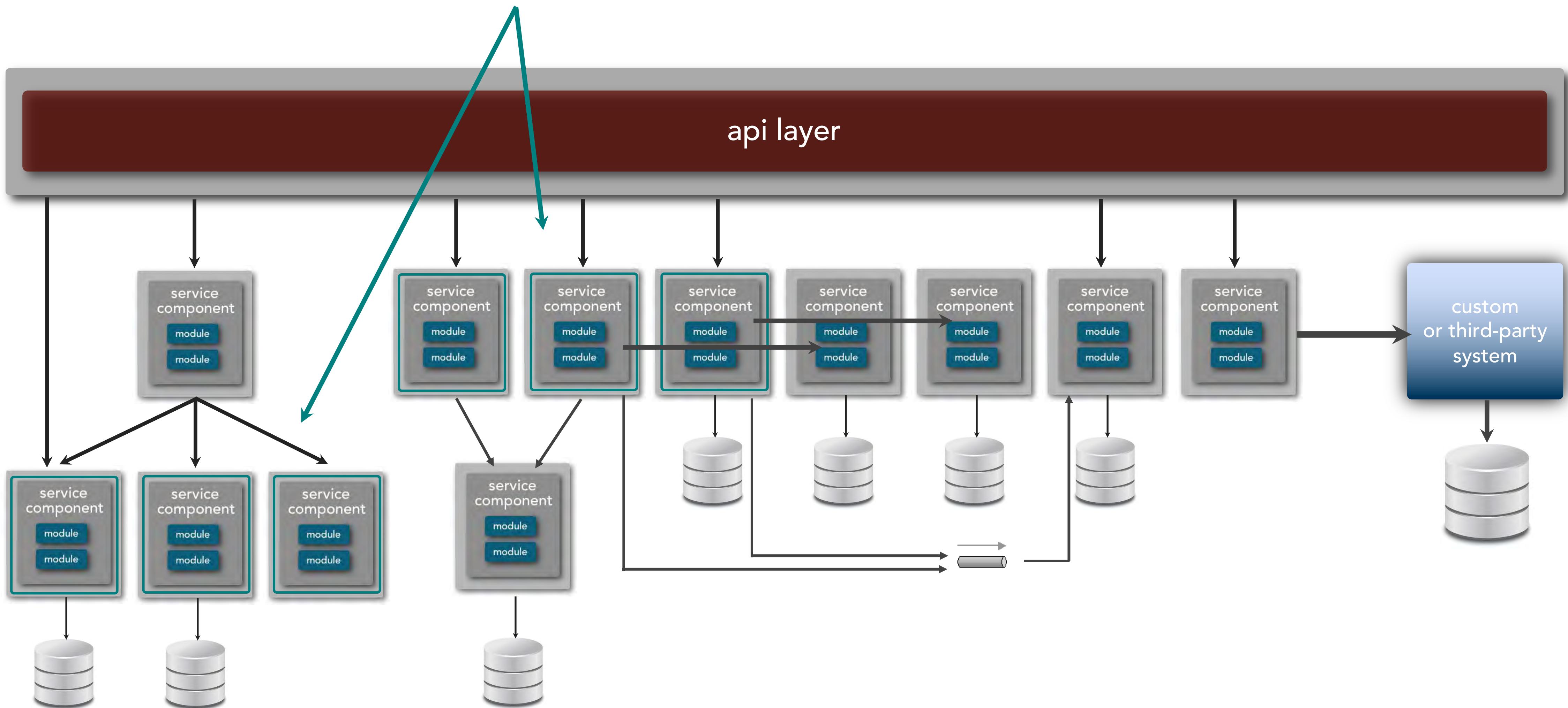
# service types

## functional services



# service types

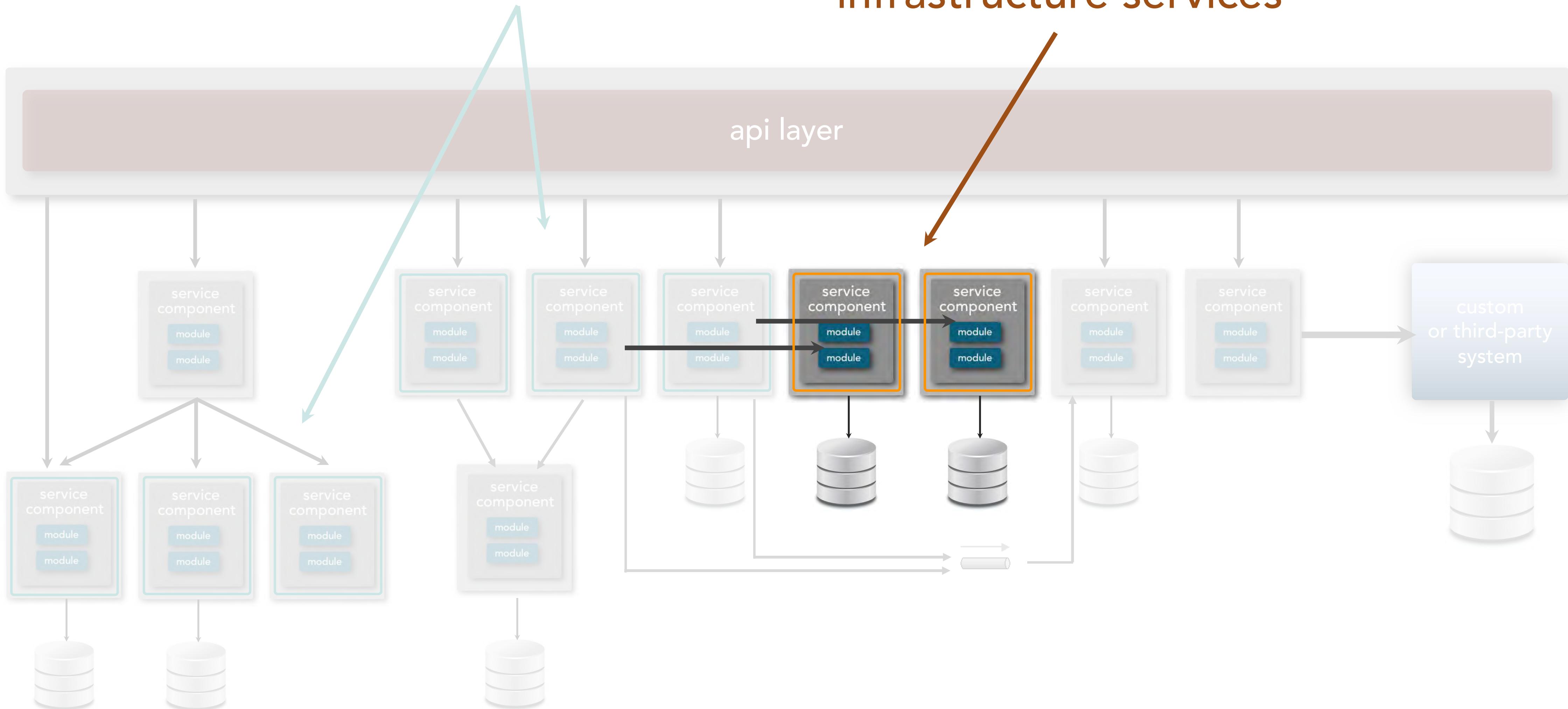
# functional services



# service types

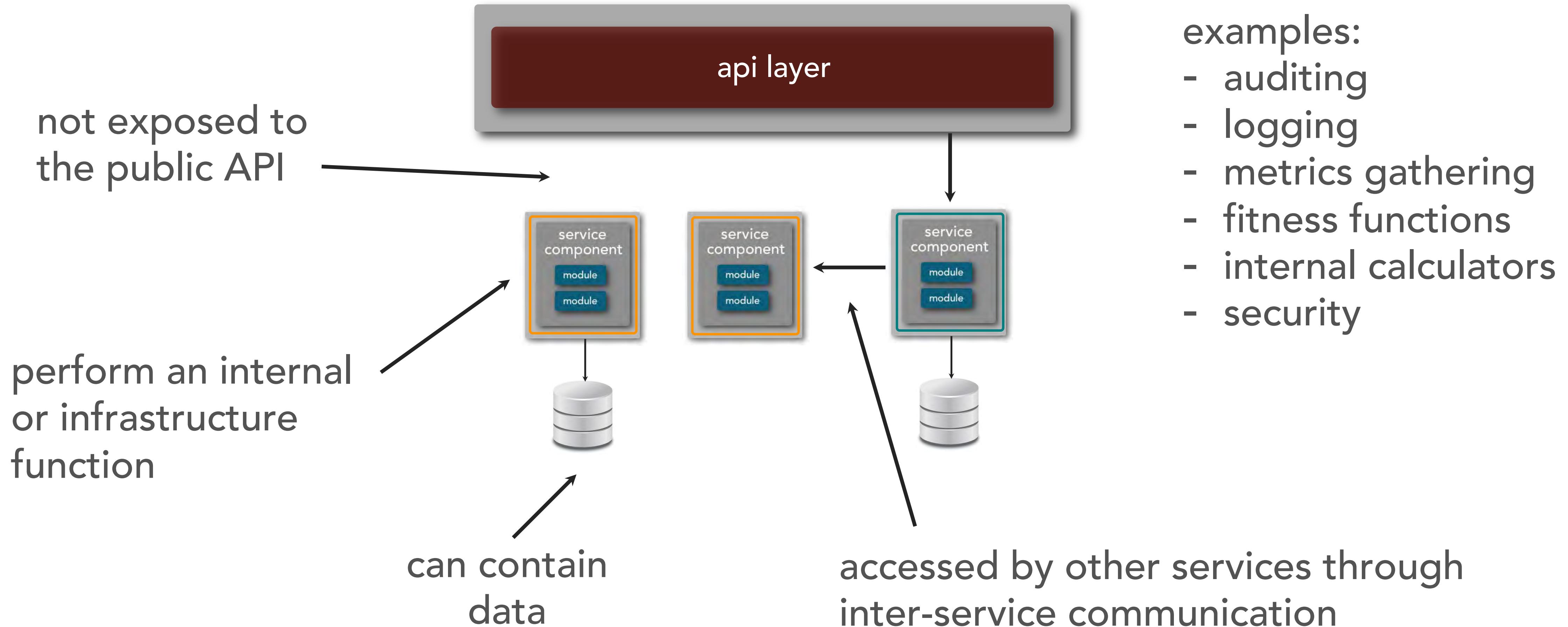
# functional services

# infrastructure services



# service types

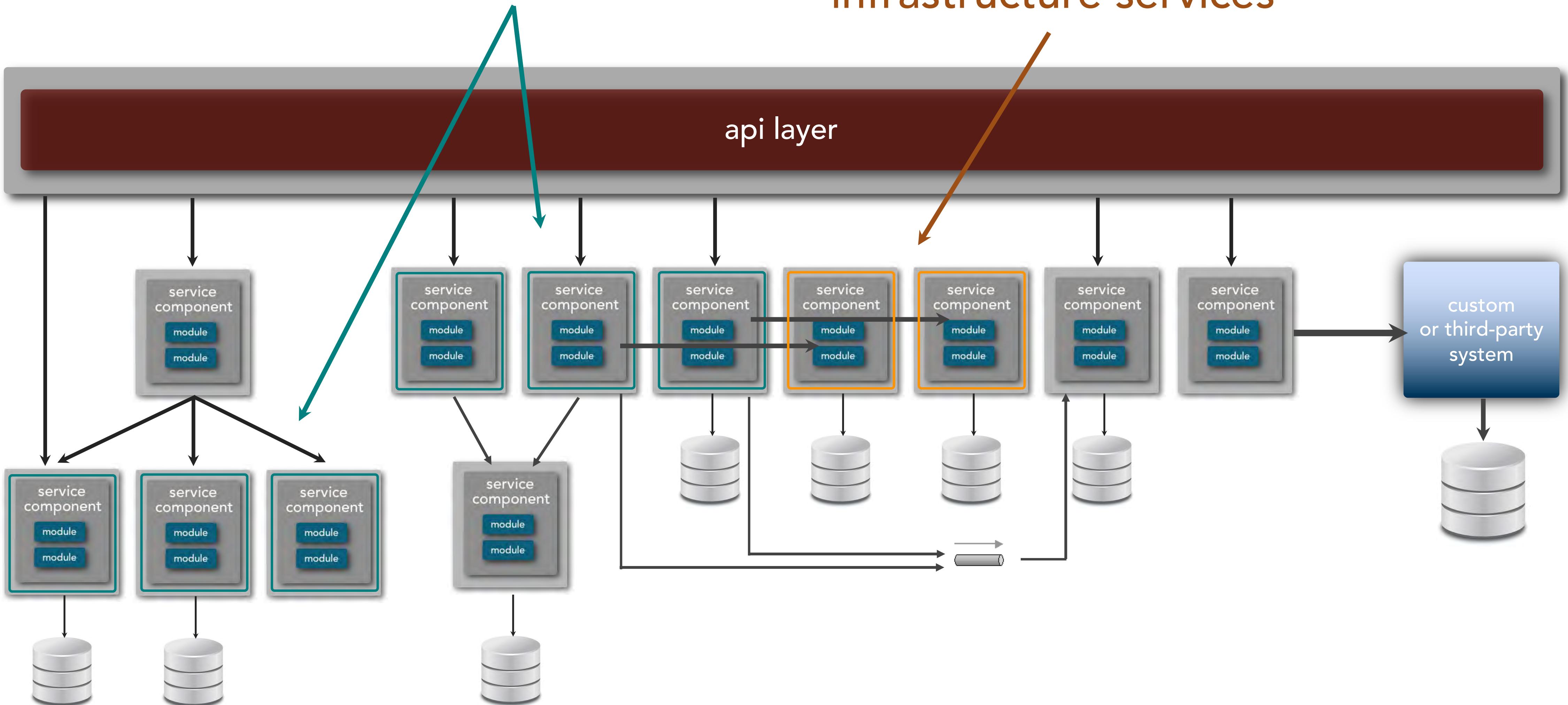
## infrastructure services



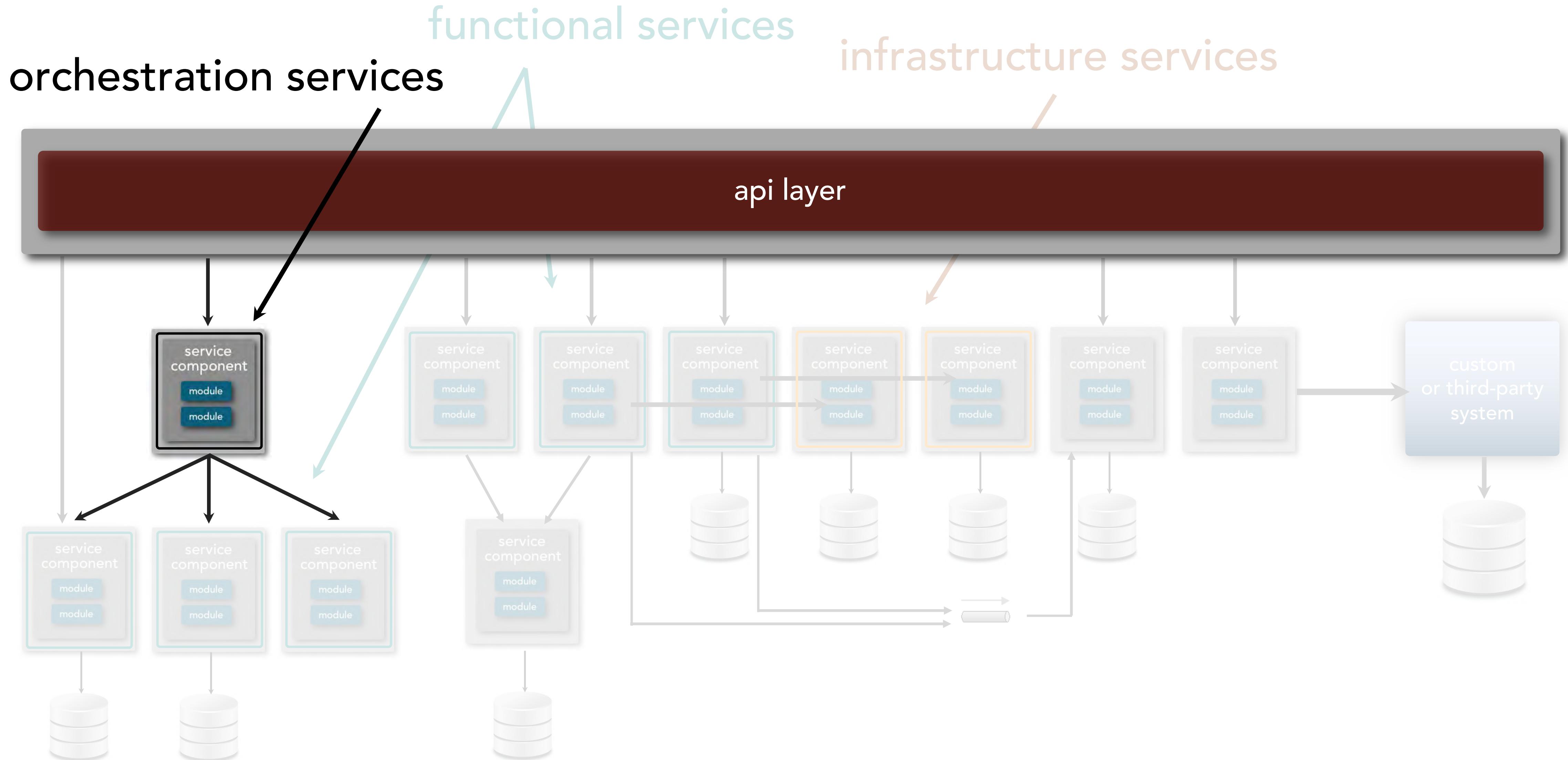
# service types

functional services

infrastructure services



# service types

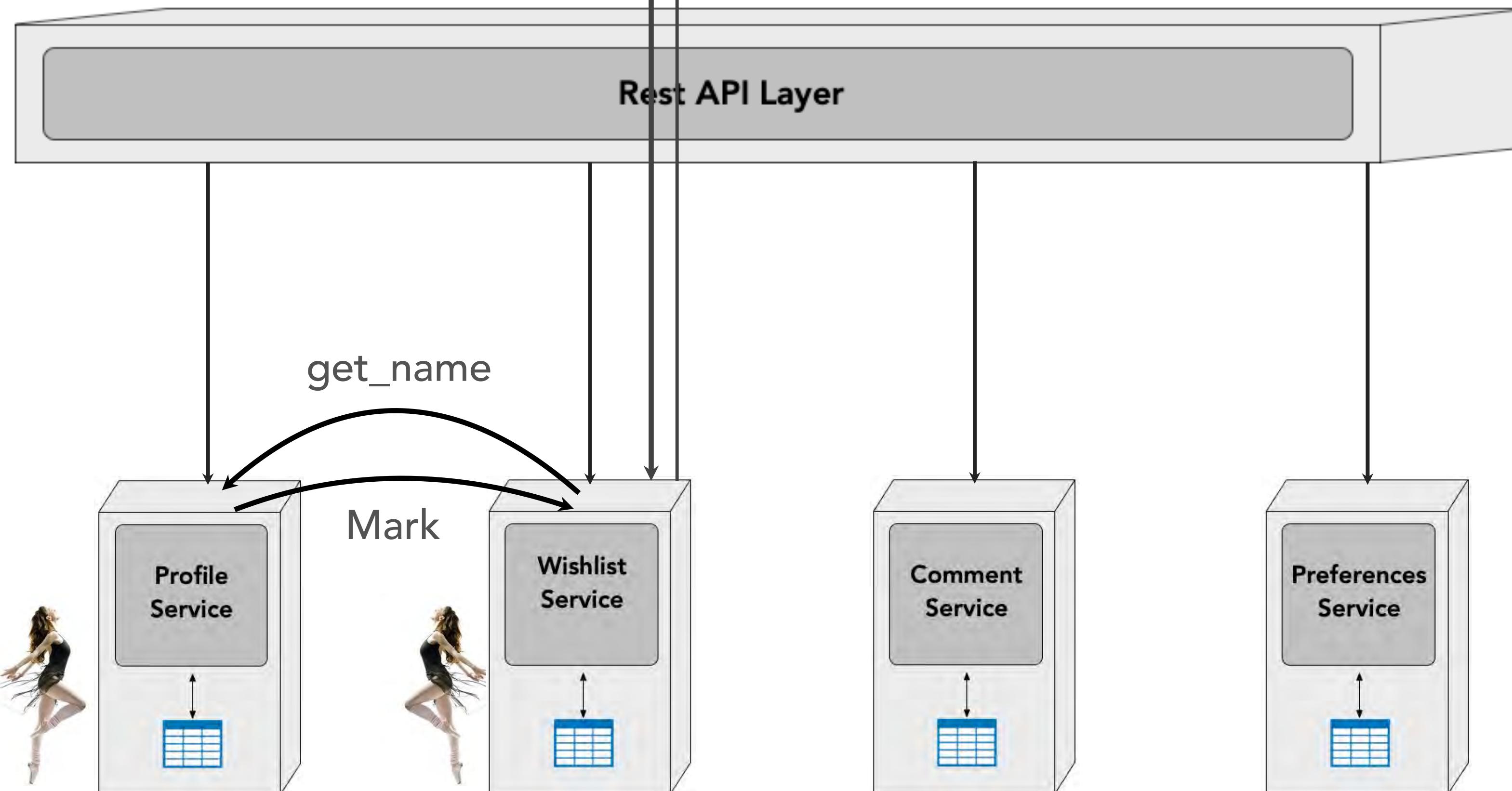


# service types

## orchestration services

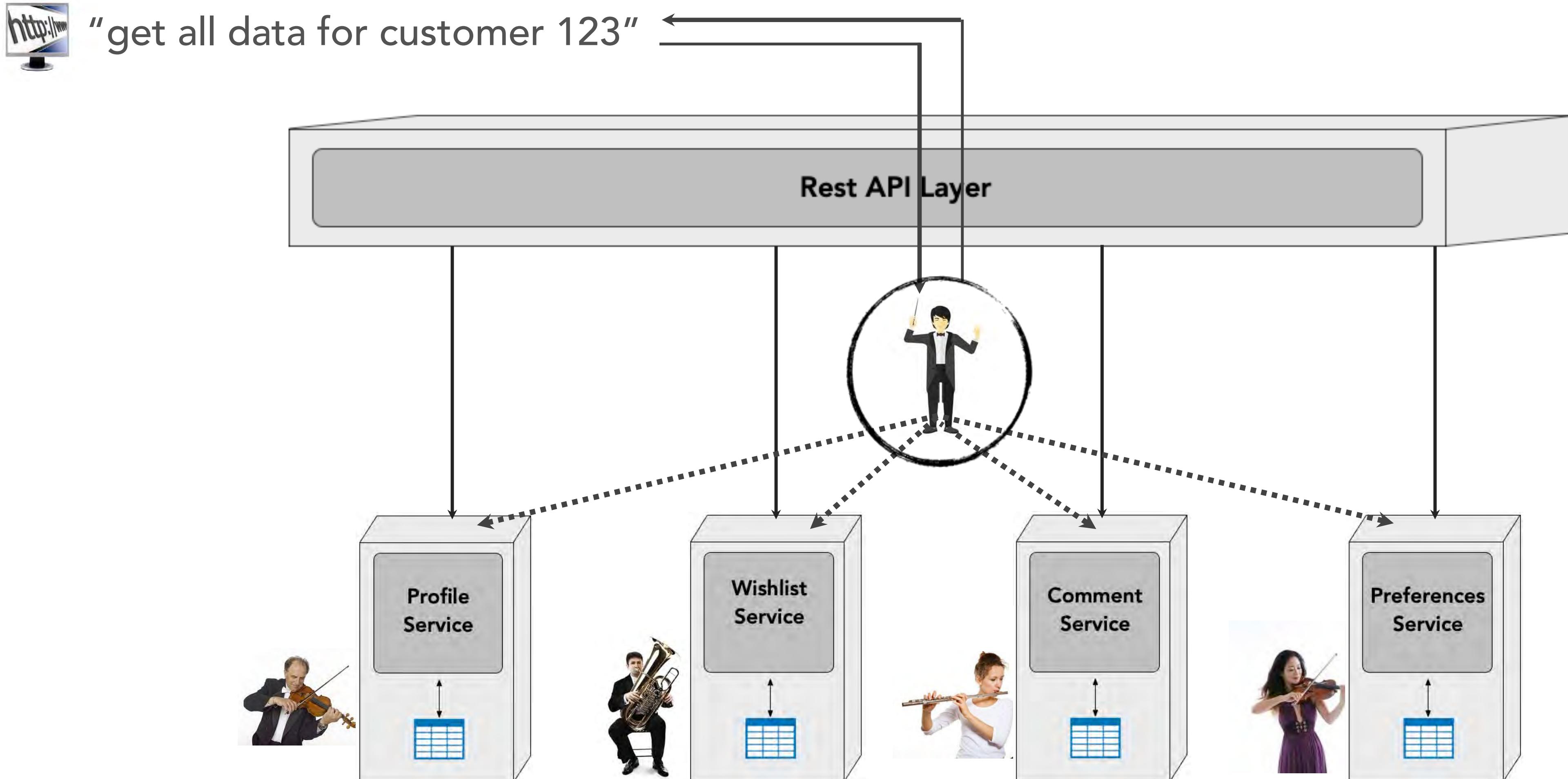


"get wishlist for customer 123"



# service types

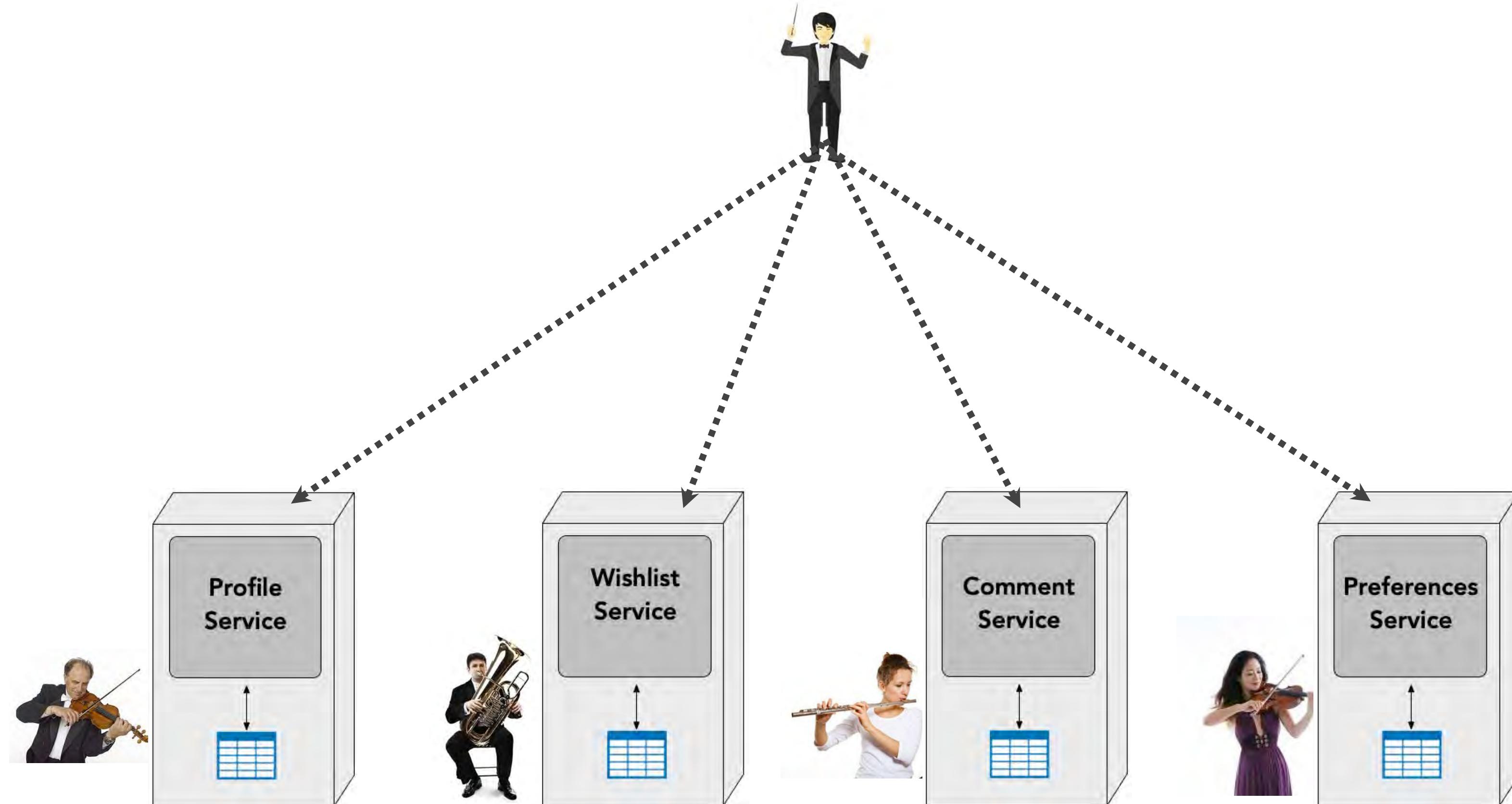
## orchestration services



# service types

## orchestration services

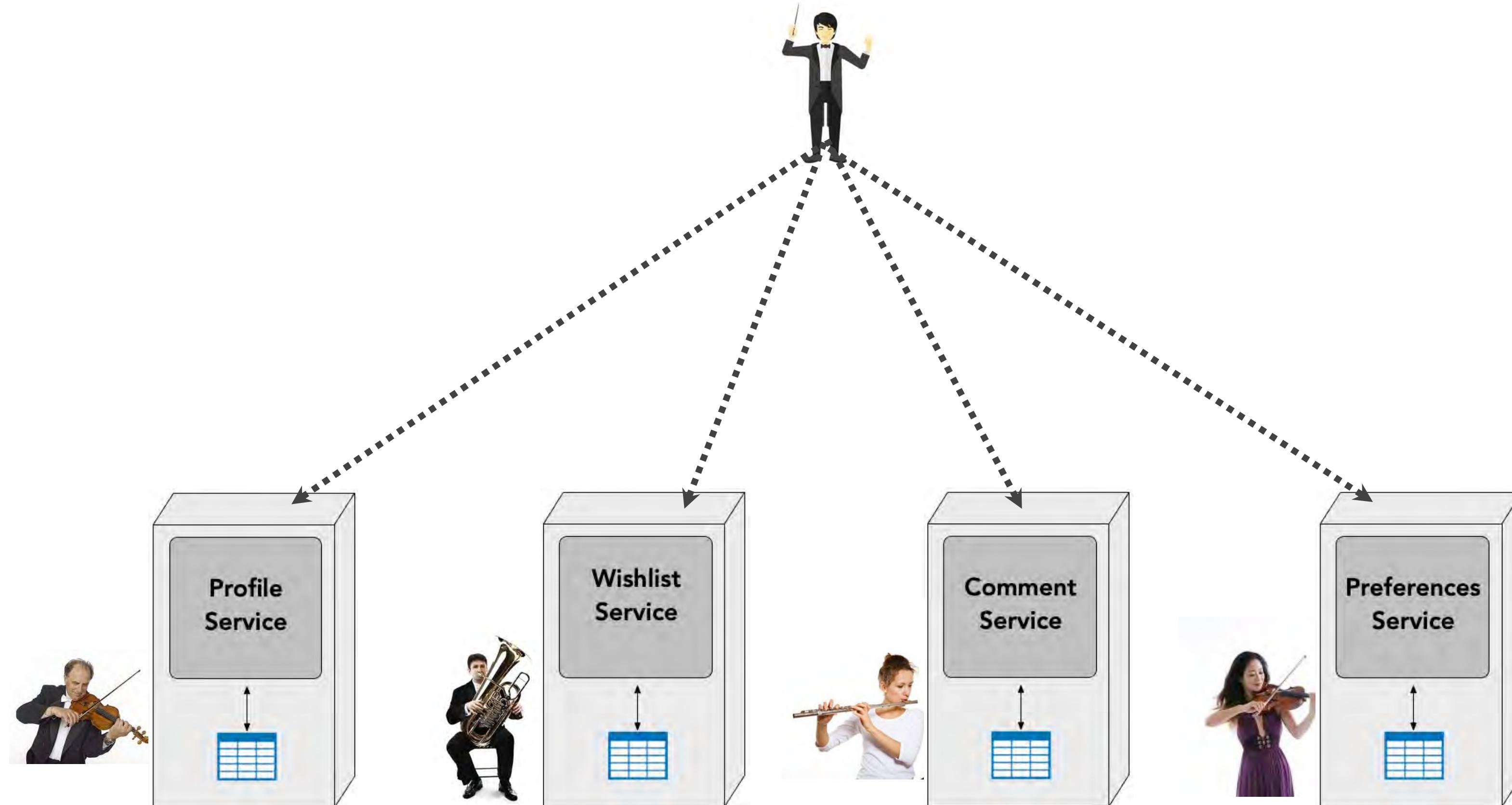
### 1. multicasting logic



# service types

## orchestration services

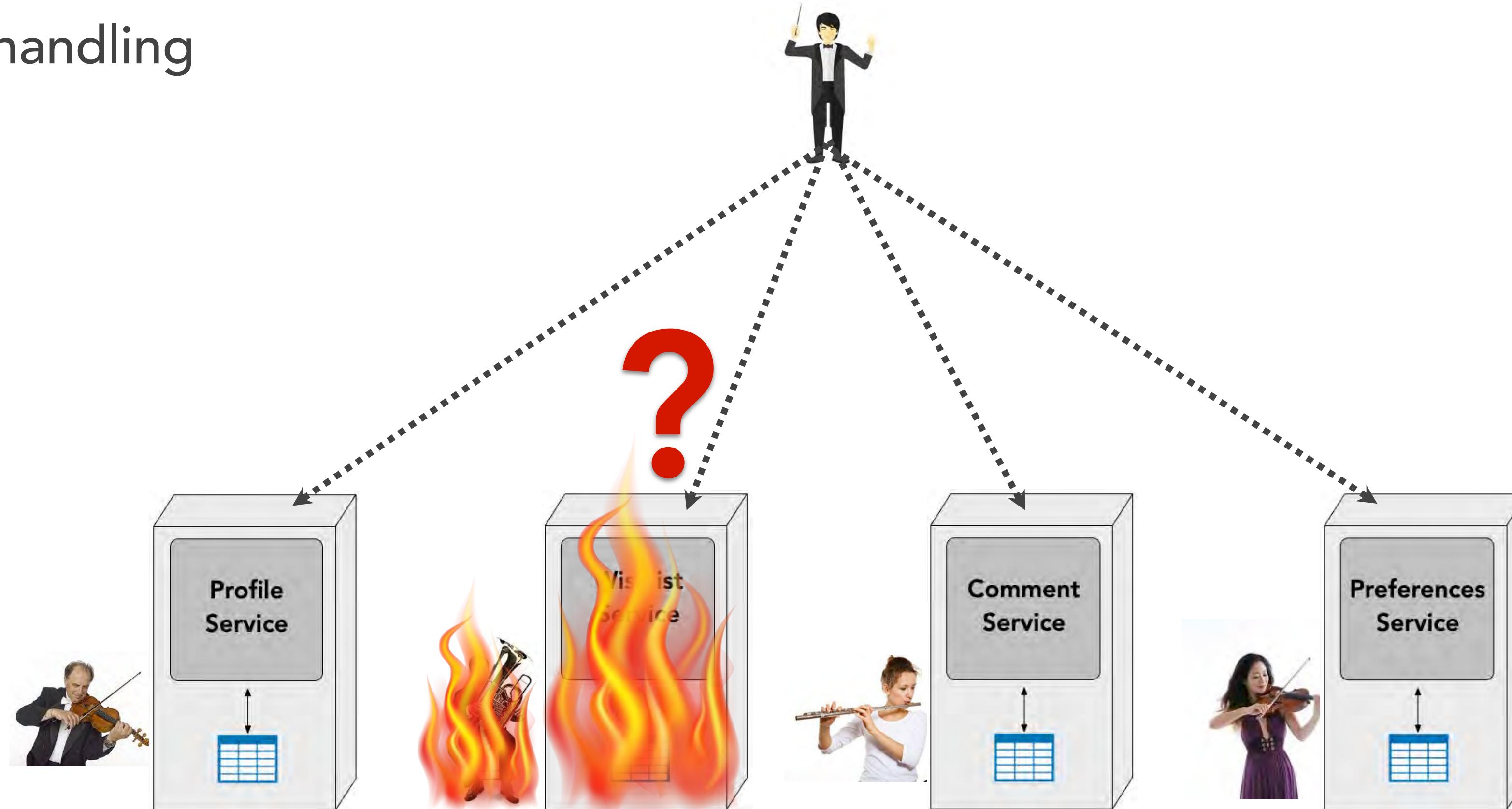
### 1. multicasting logic



# service types

## orchestration services

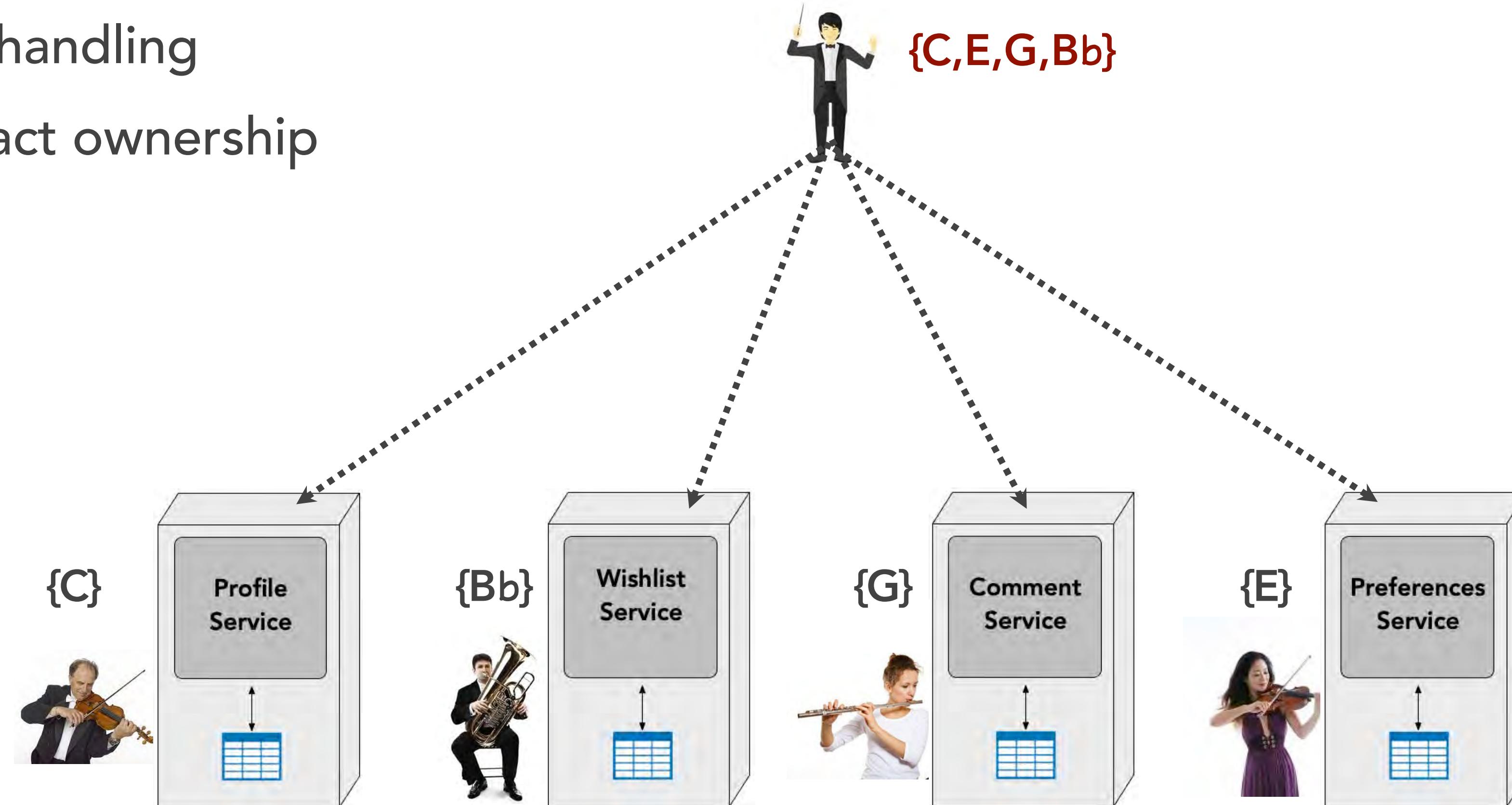
1. multicasting logic
2. error handling



# service types

## orchestration services

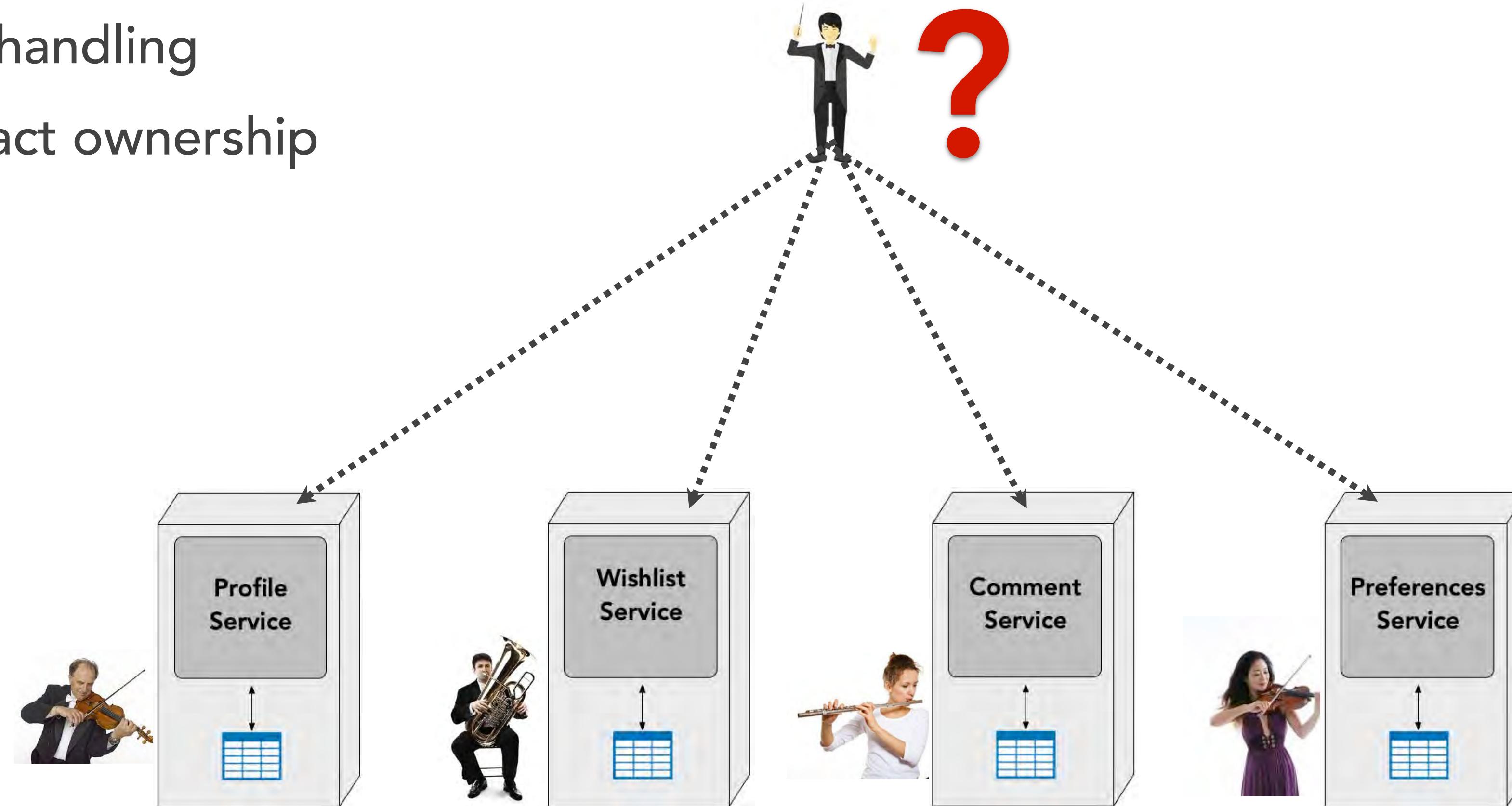
1. multicasting logic
2. error handling
3. contract ownership



# service types

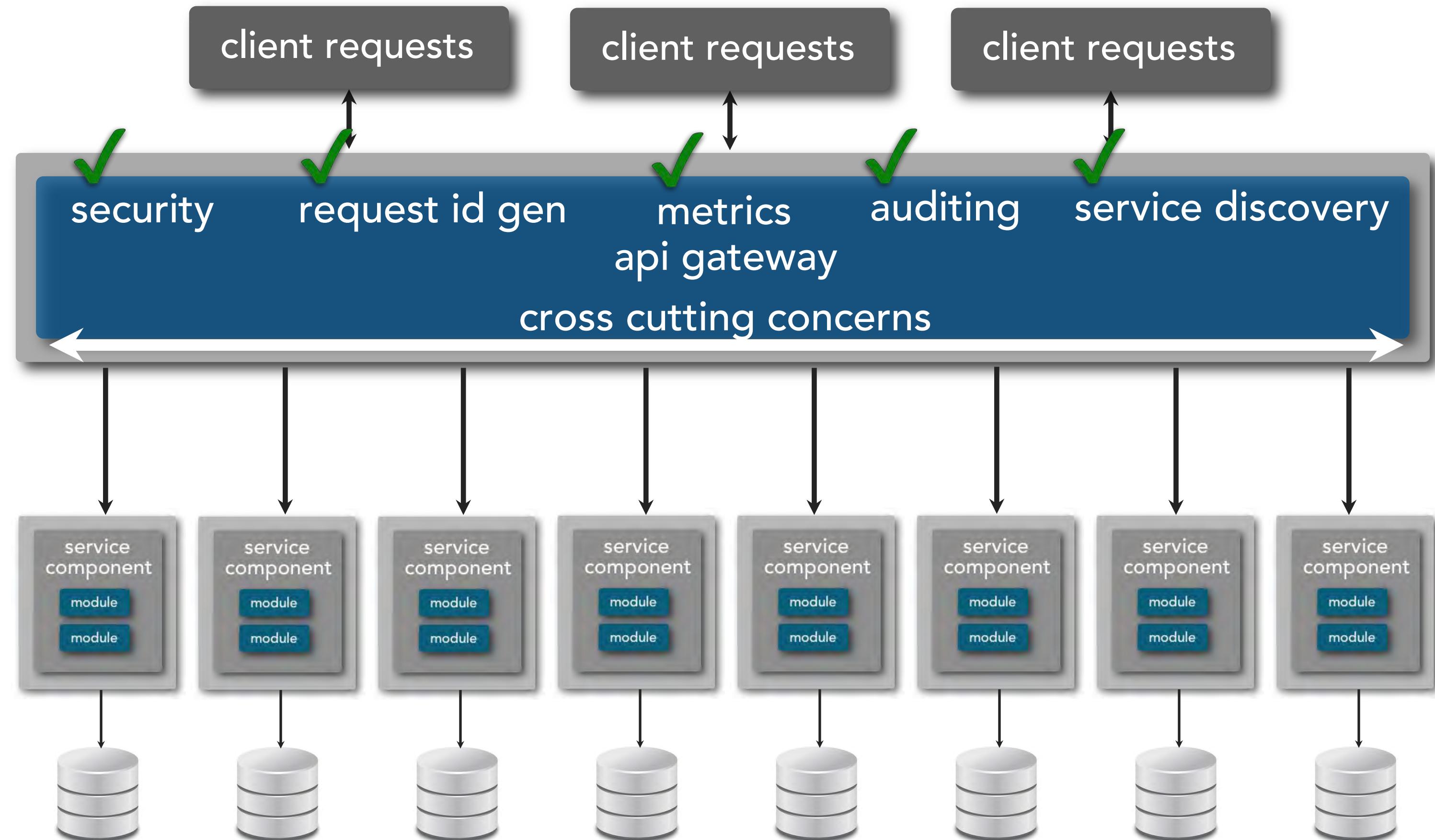
## orchestration services

1. multicasting logic
2. error handling
3. contract ownership



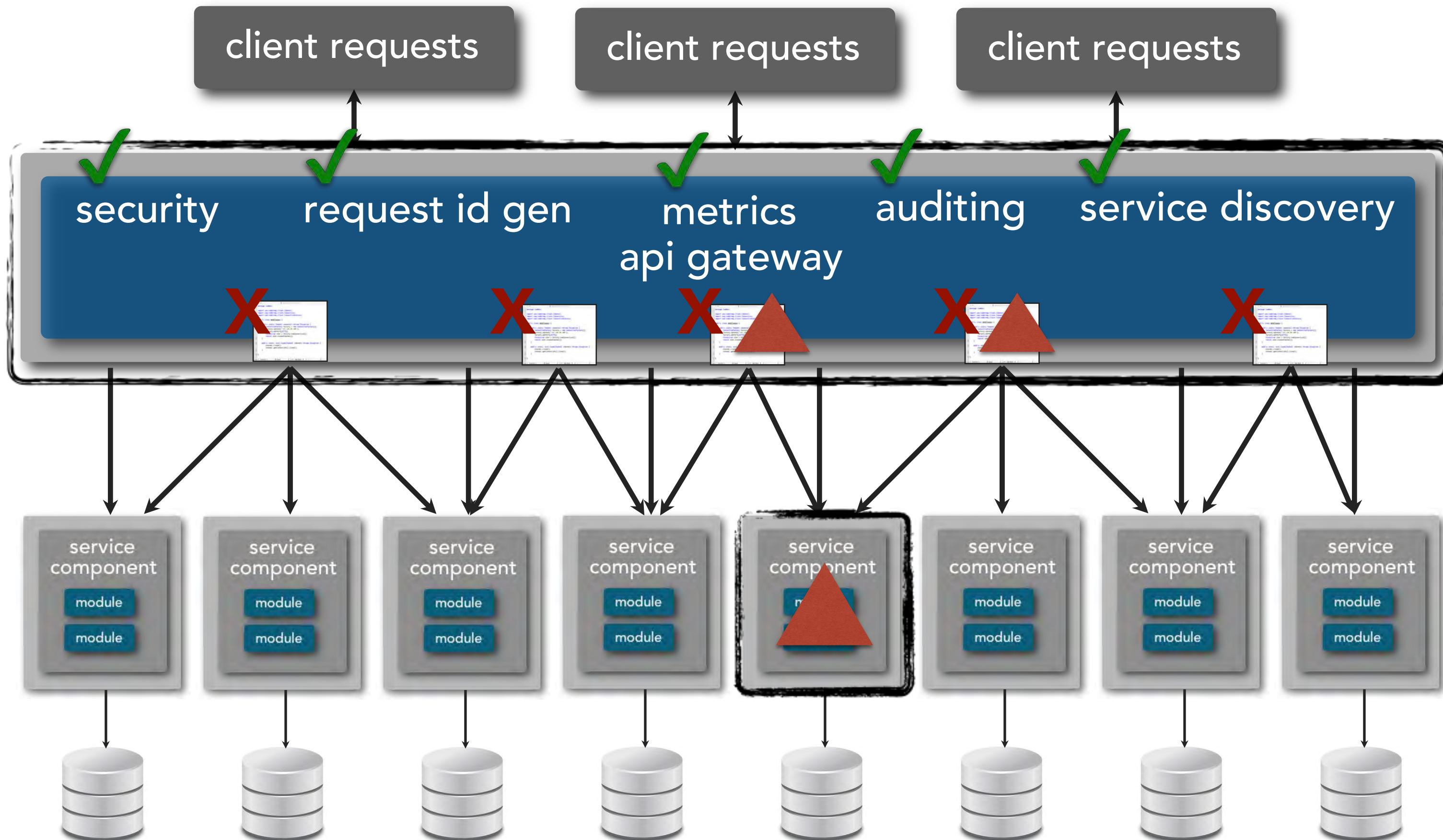
# service types

## orchestration services



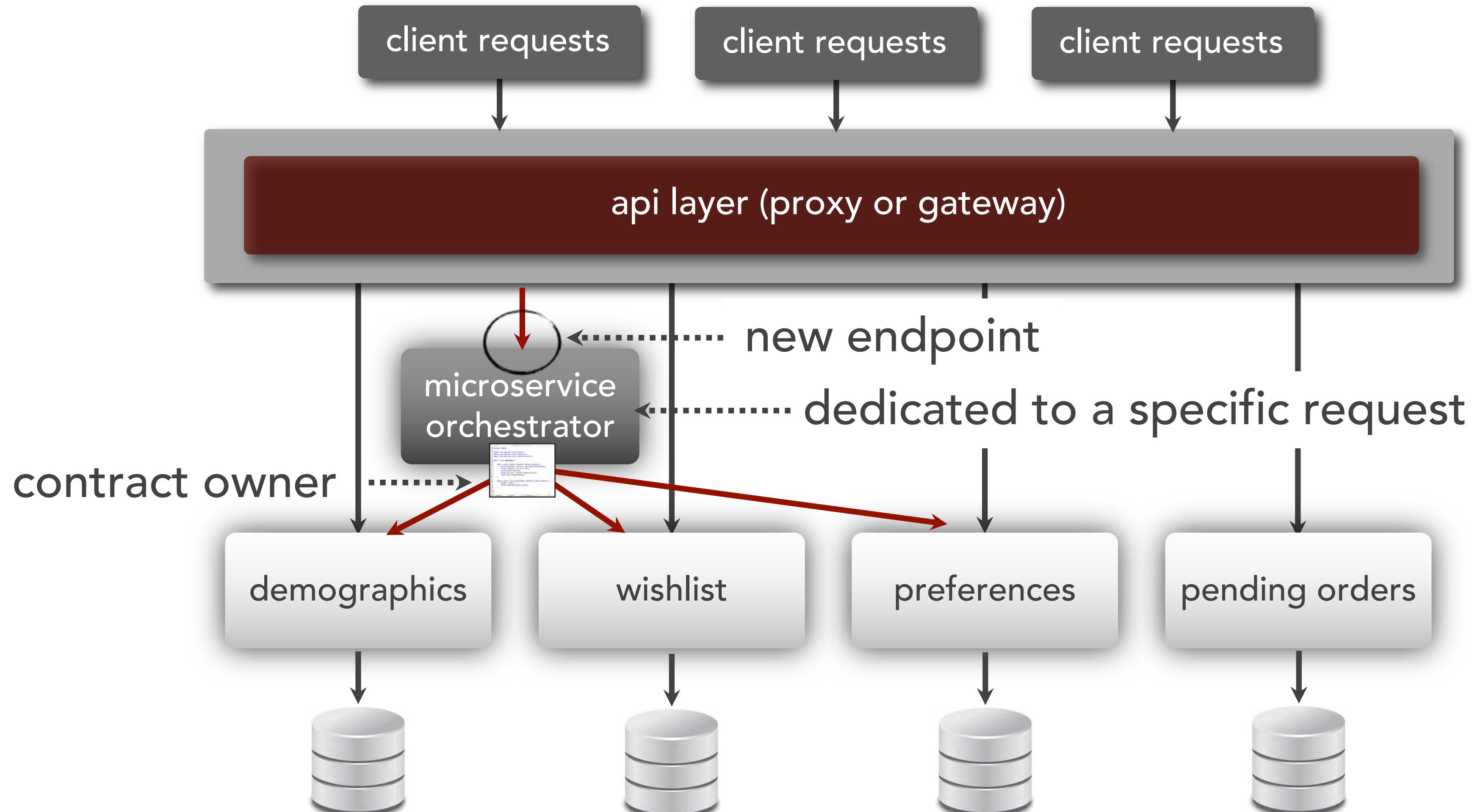
# service types

## orchestration services

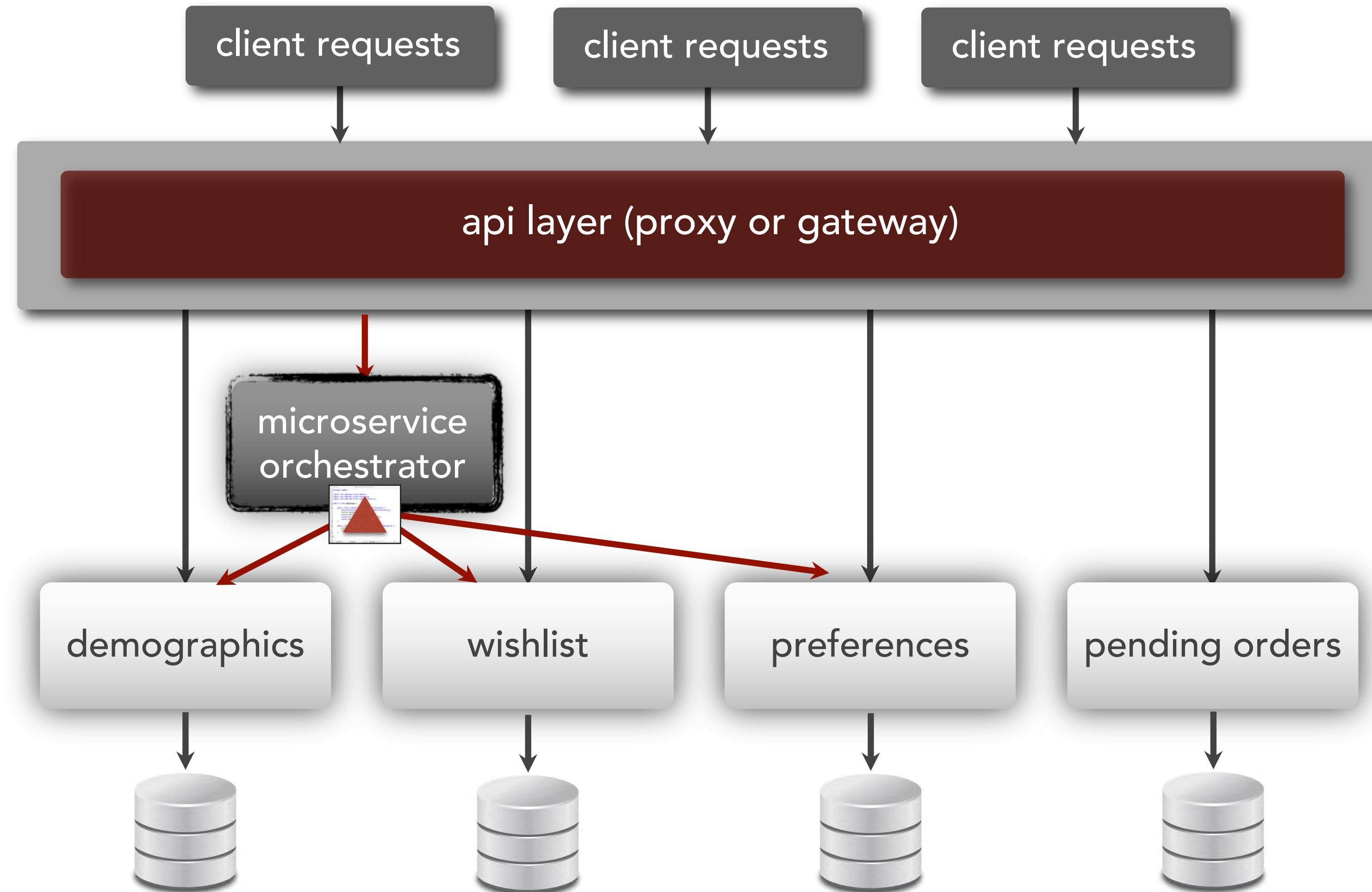


# service types

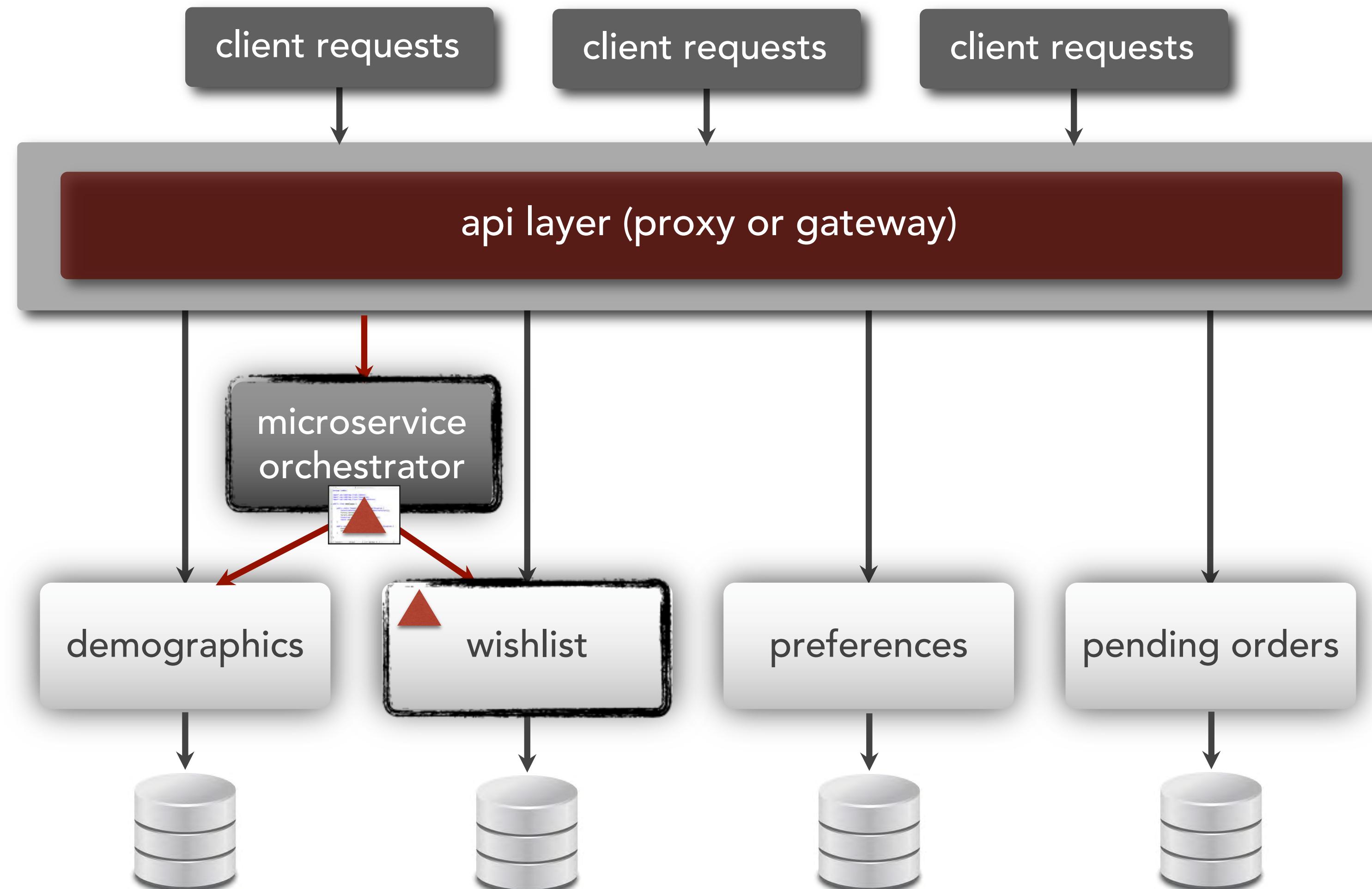
## orchestration services



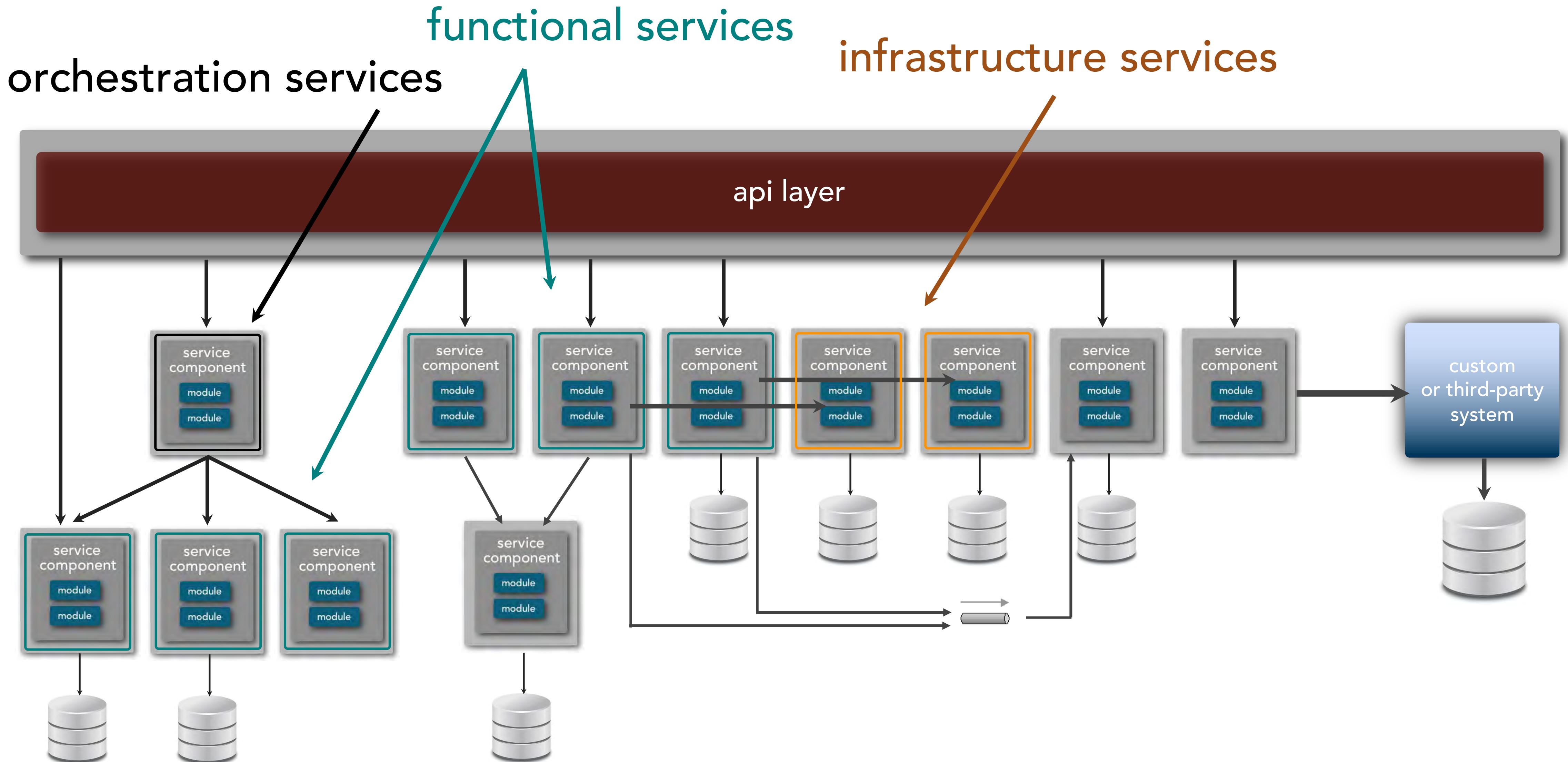
# service types orchestration services



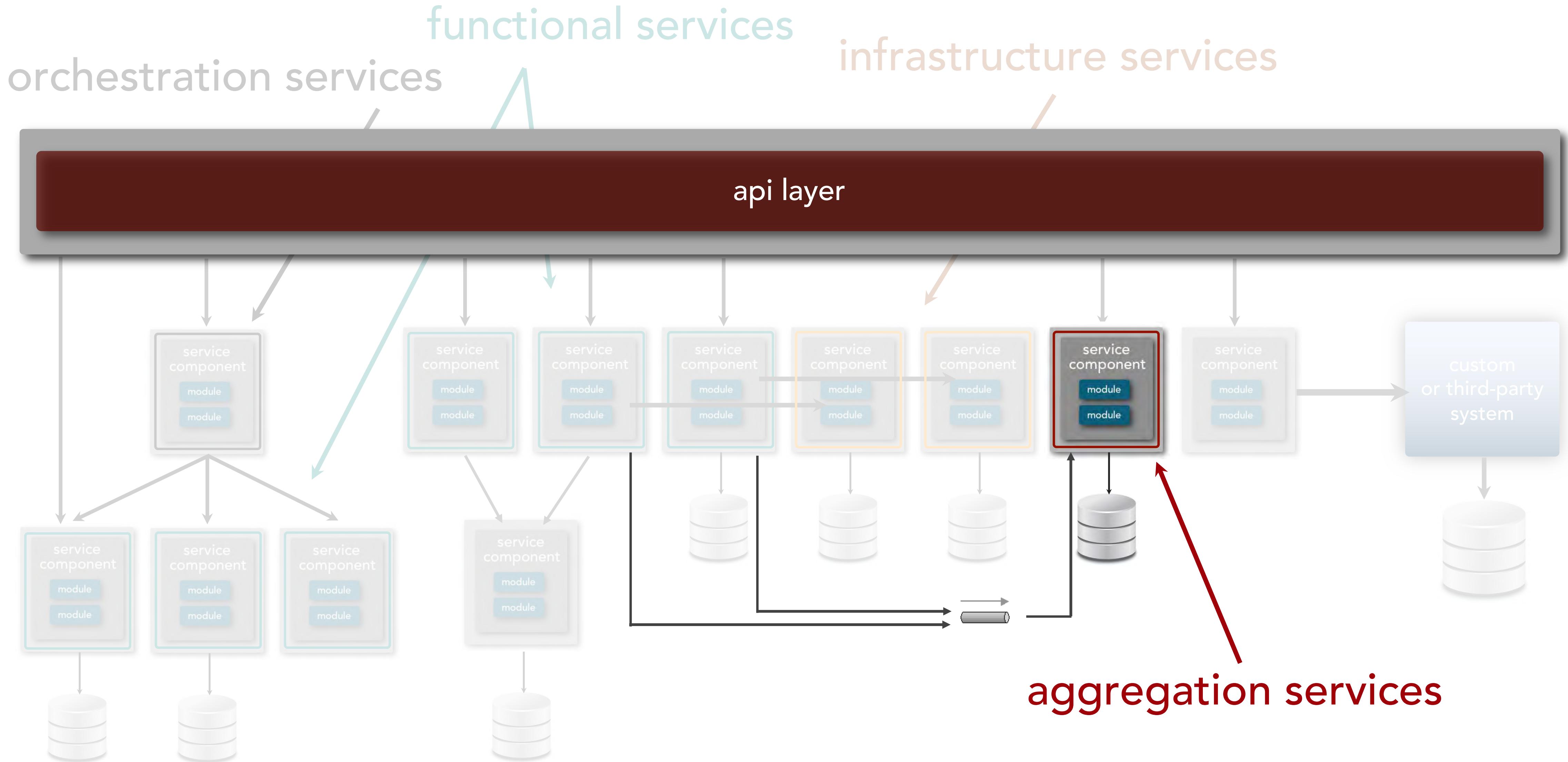
# service types orchestration services



# service types

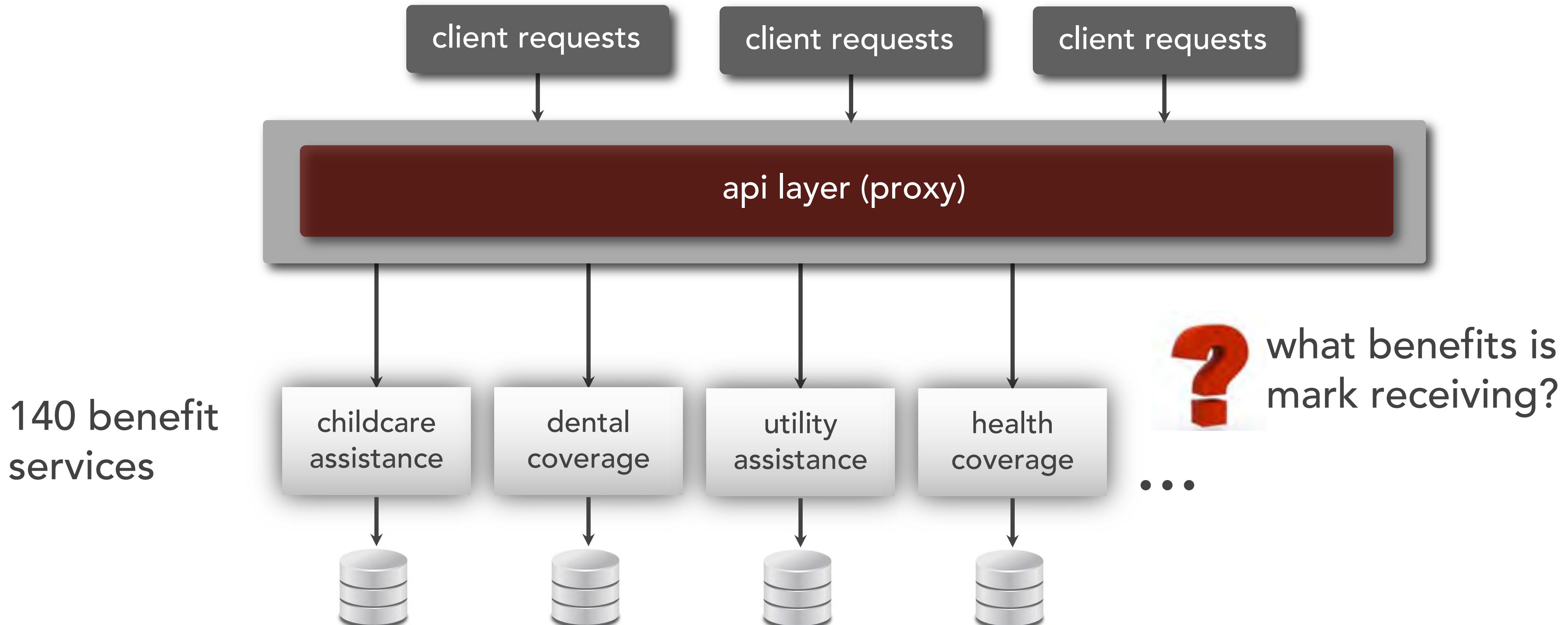


# service types



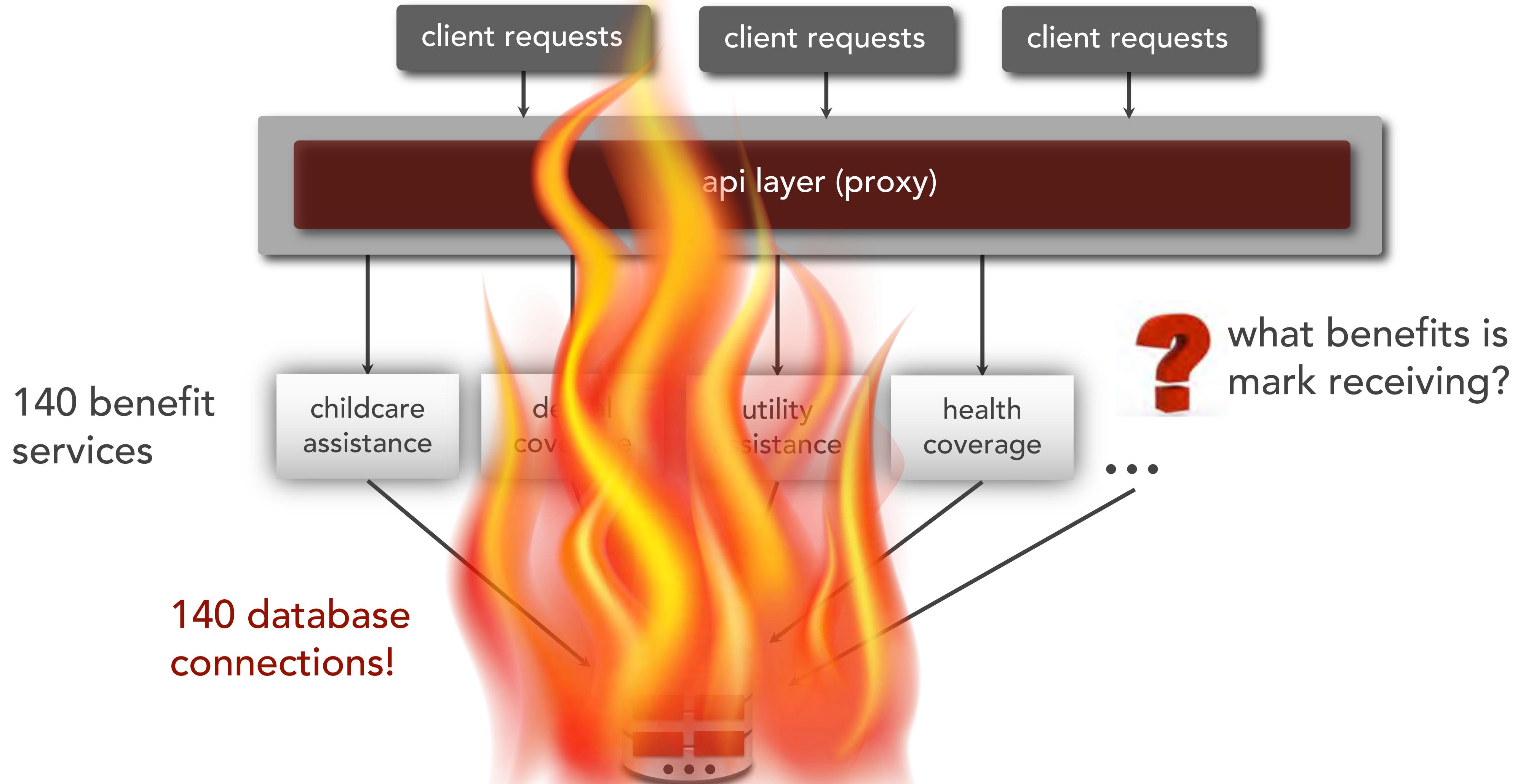
# service types

## aggregation services



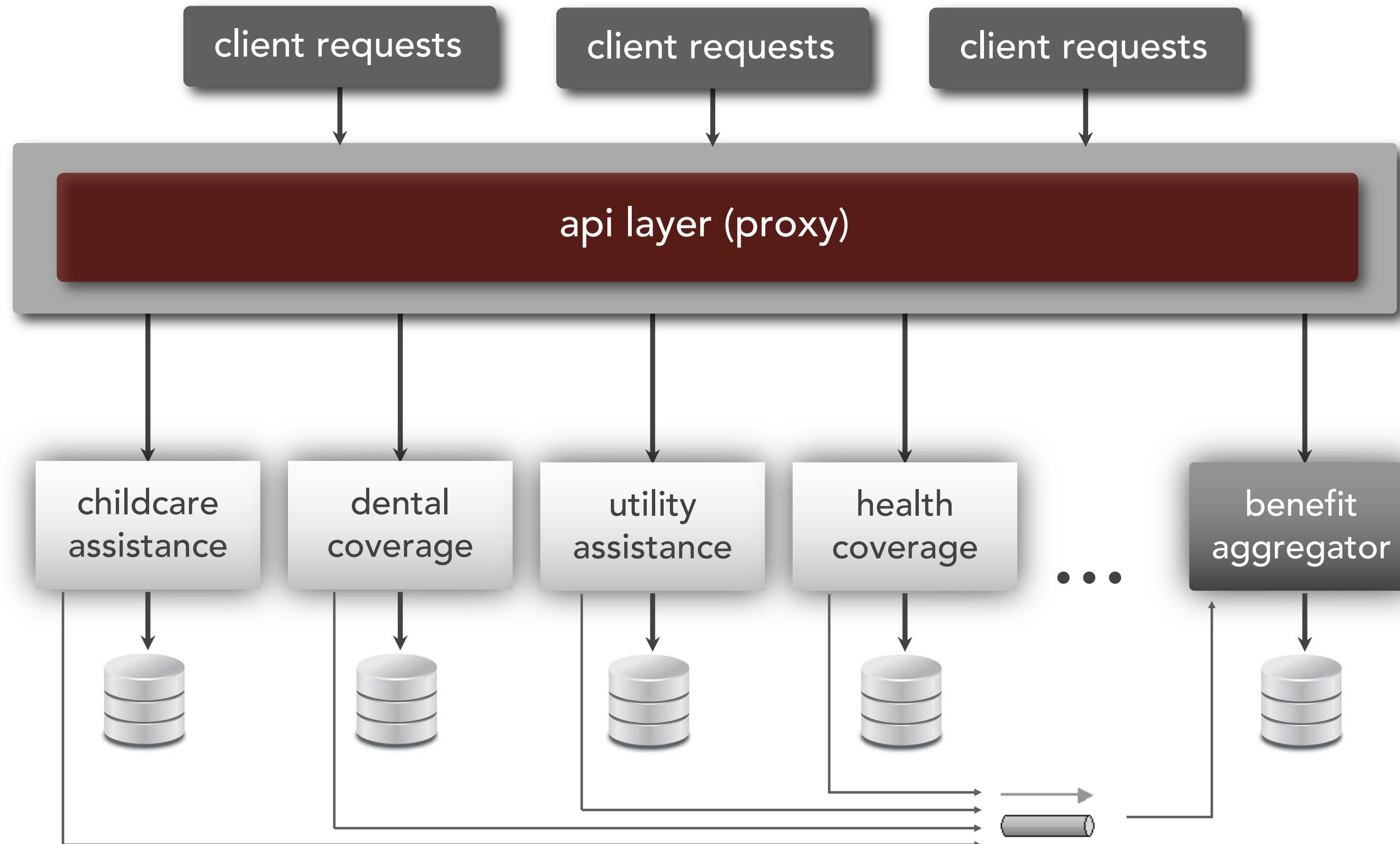
# service types

## aggregation services



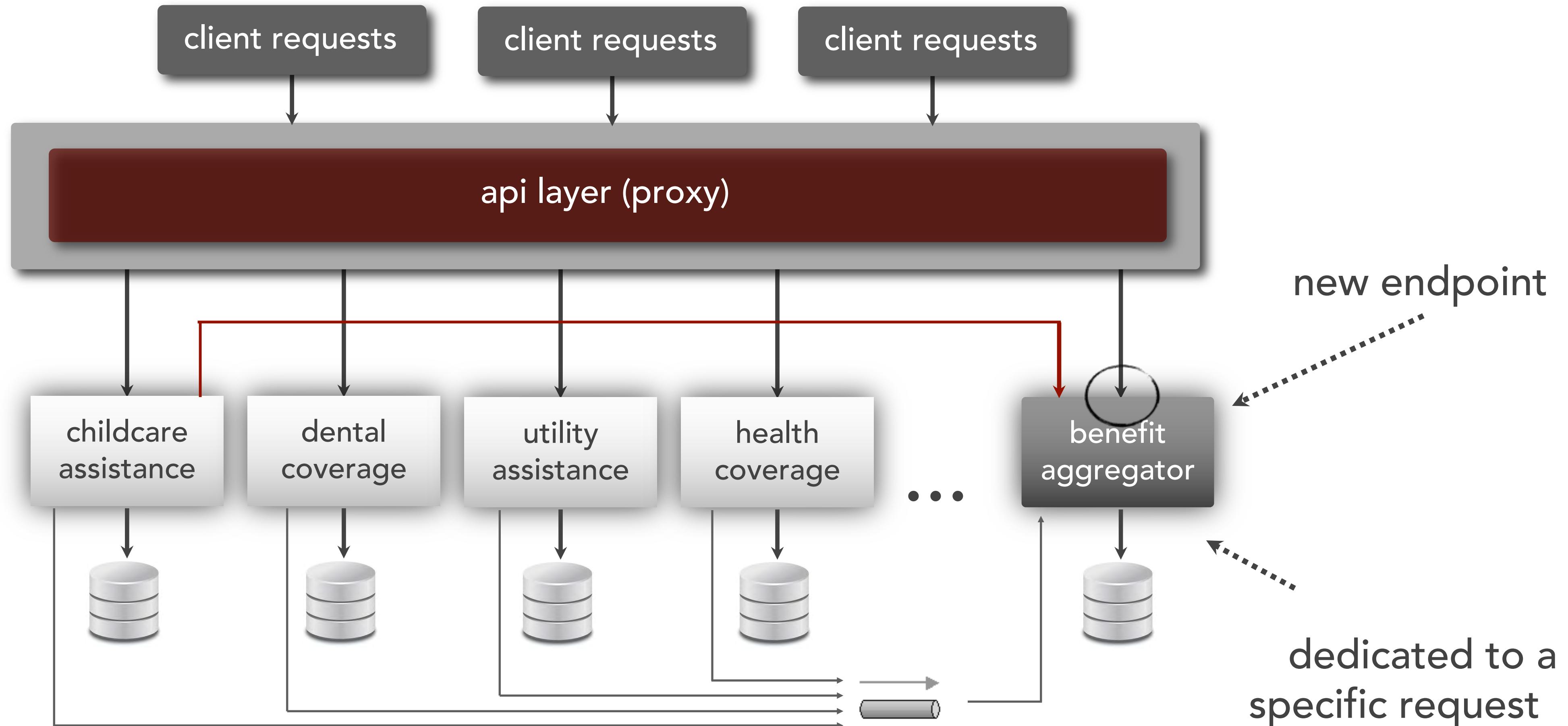
# service types

## aggregation services

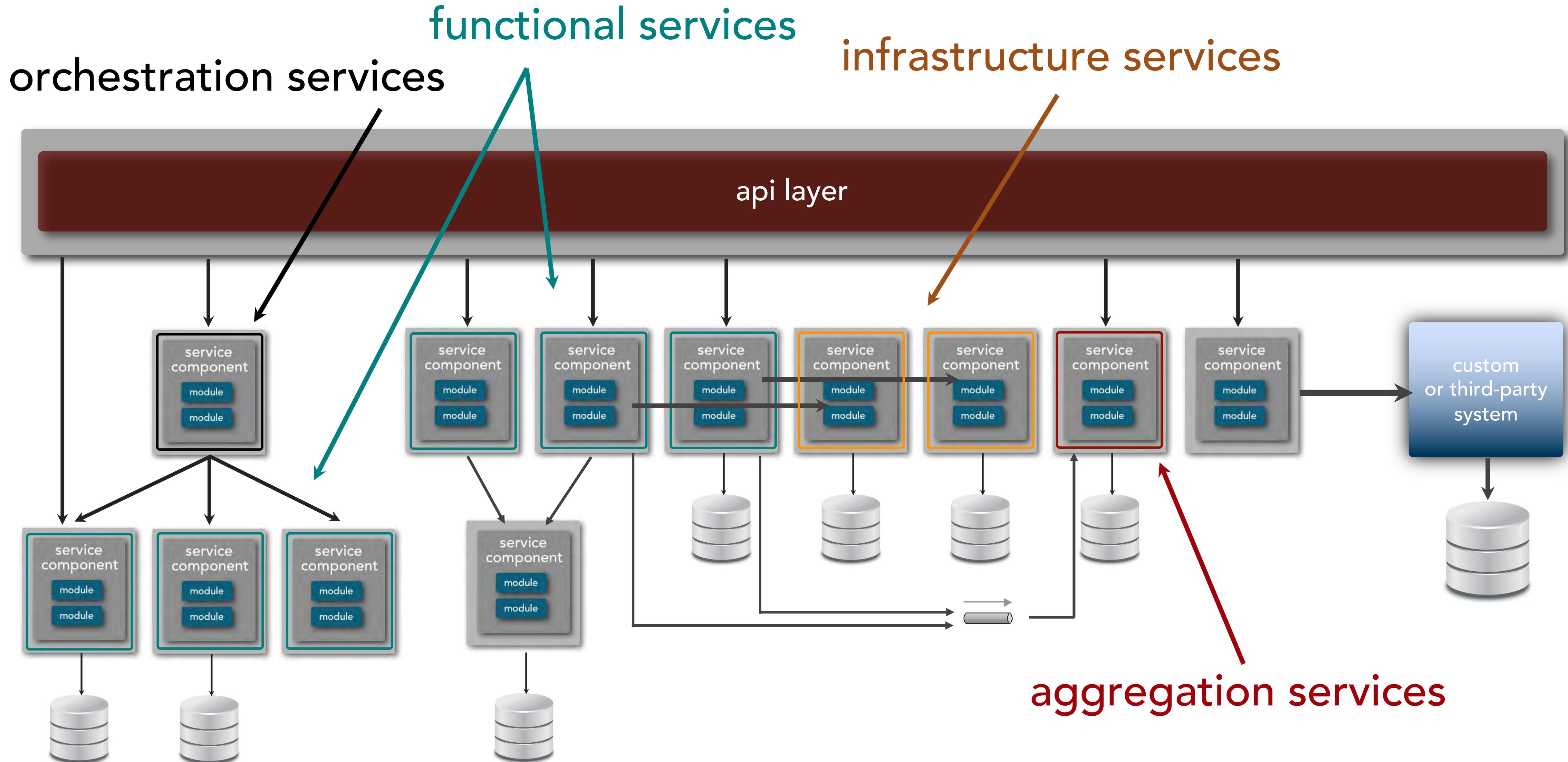


# service types

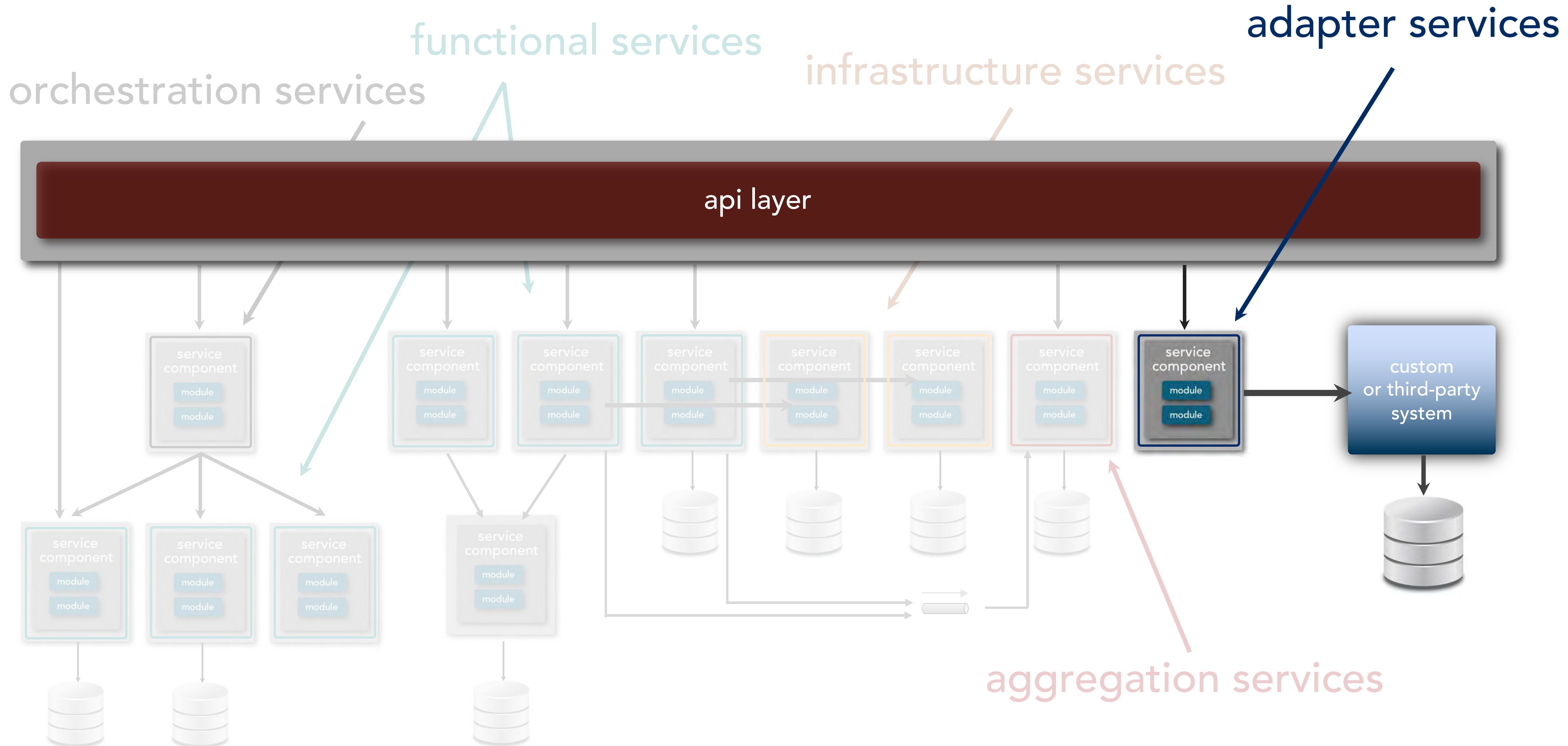
## aggregation services



# service types

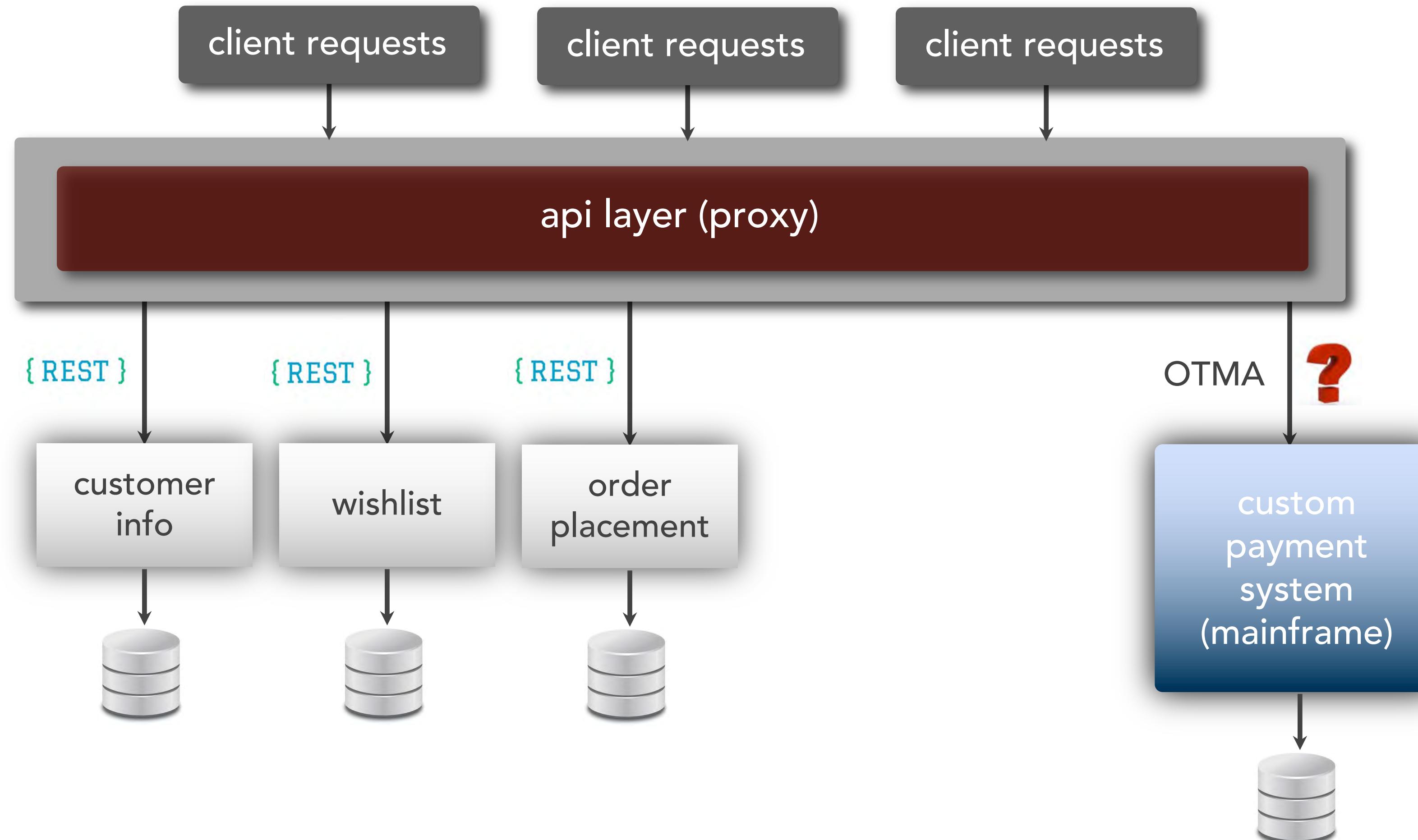


# service types



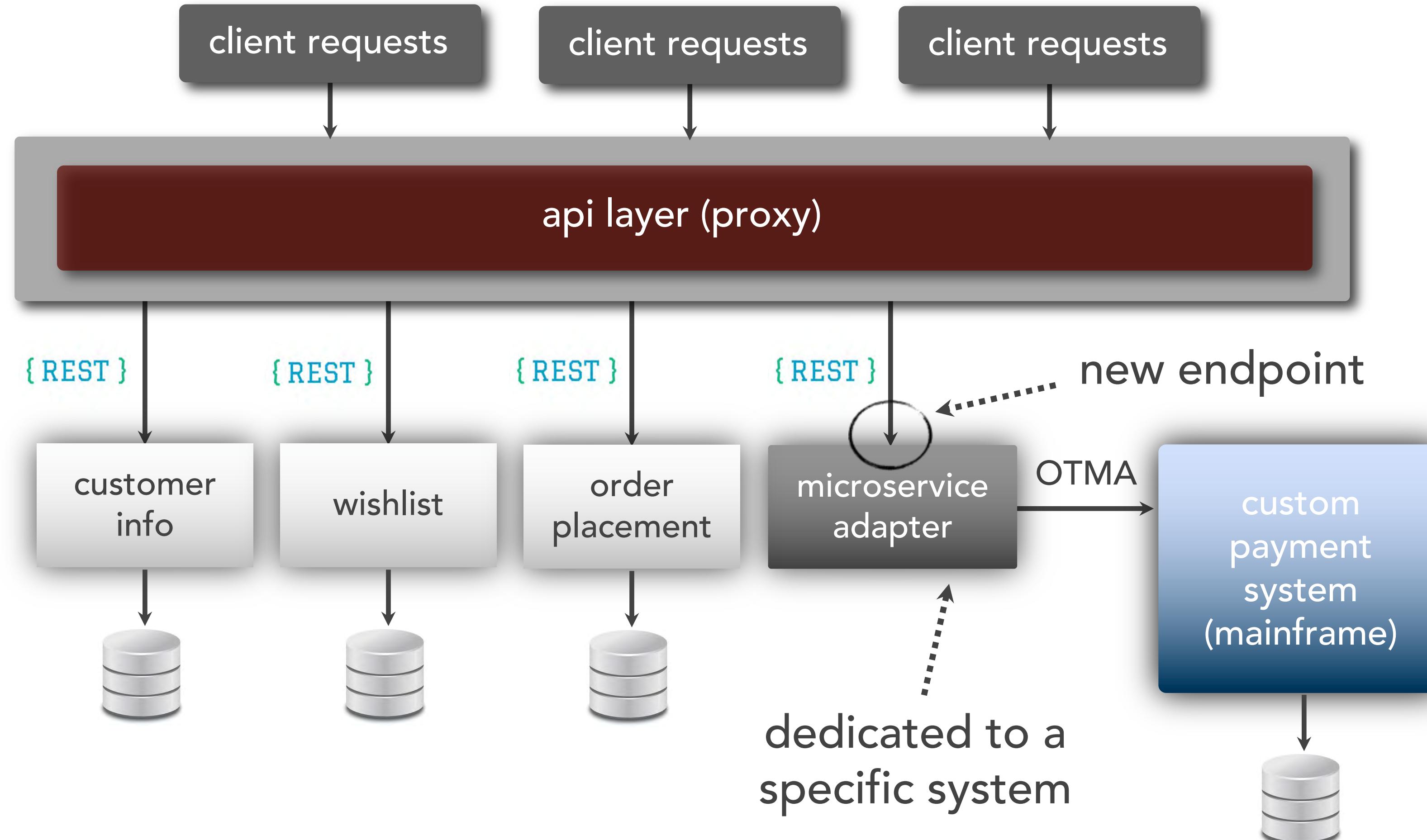
# service types

## adapter services



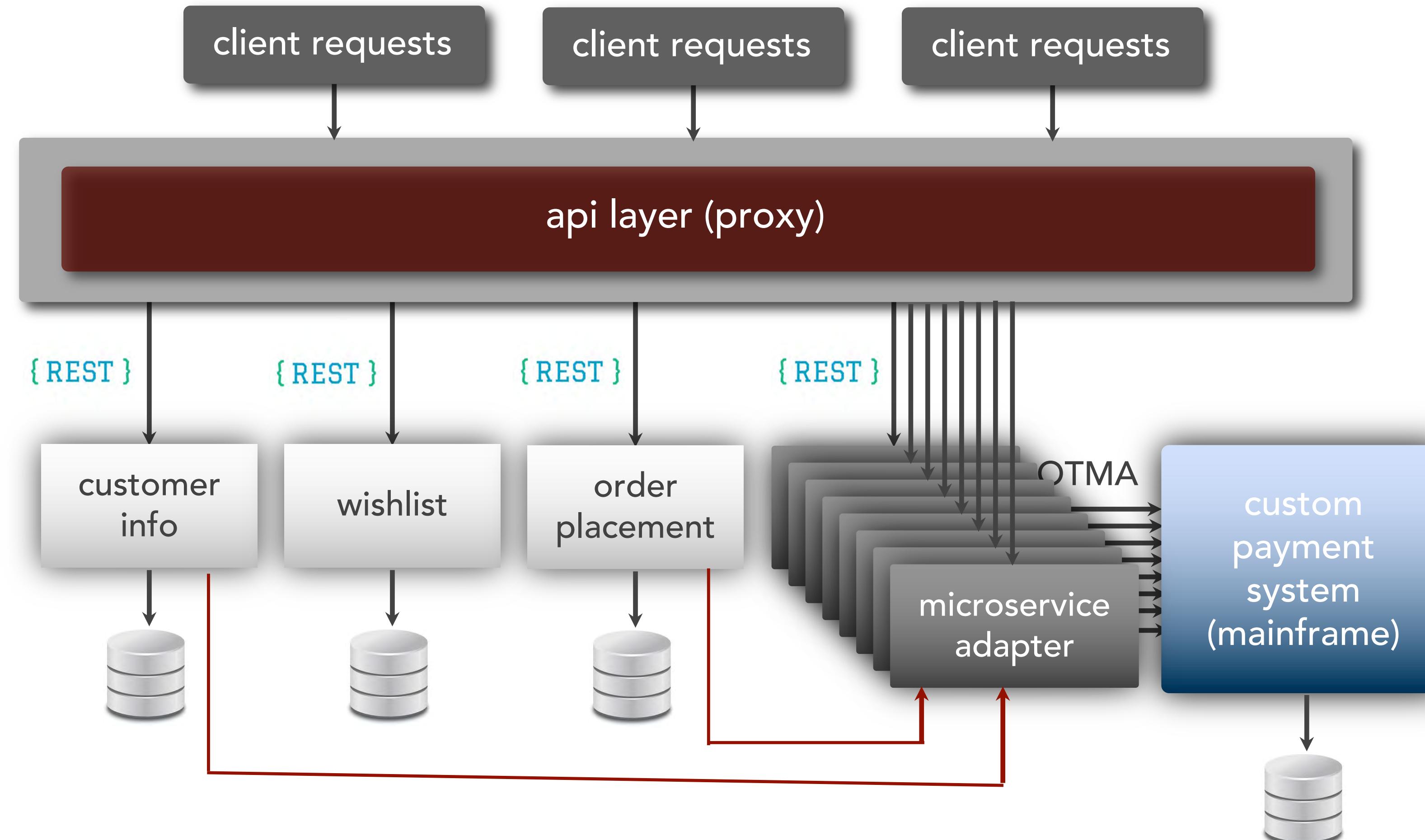
# service types

## adapter services



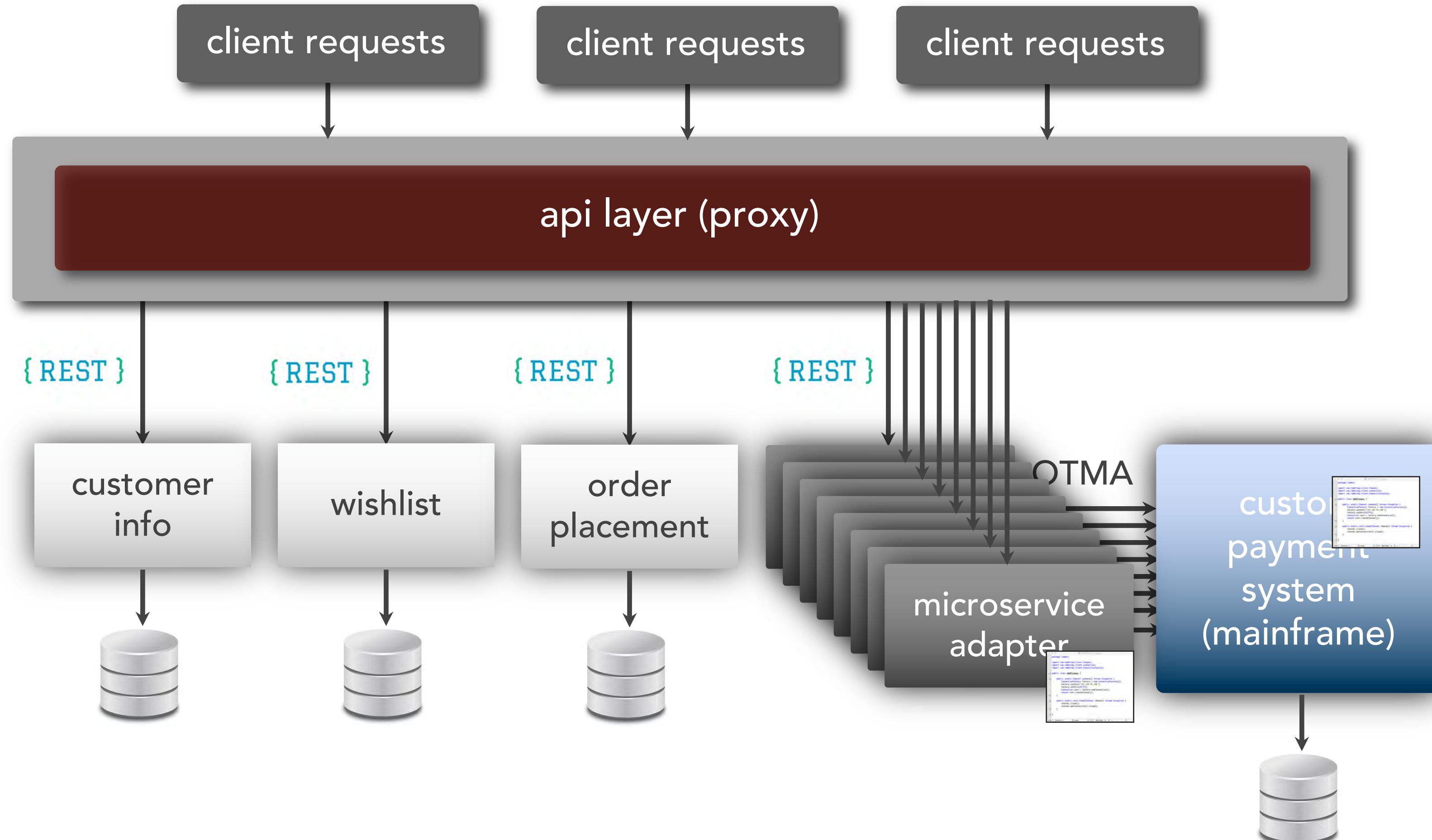
# service types

## adapter services



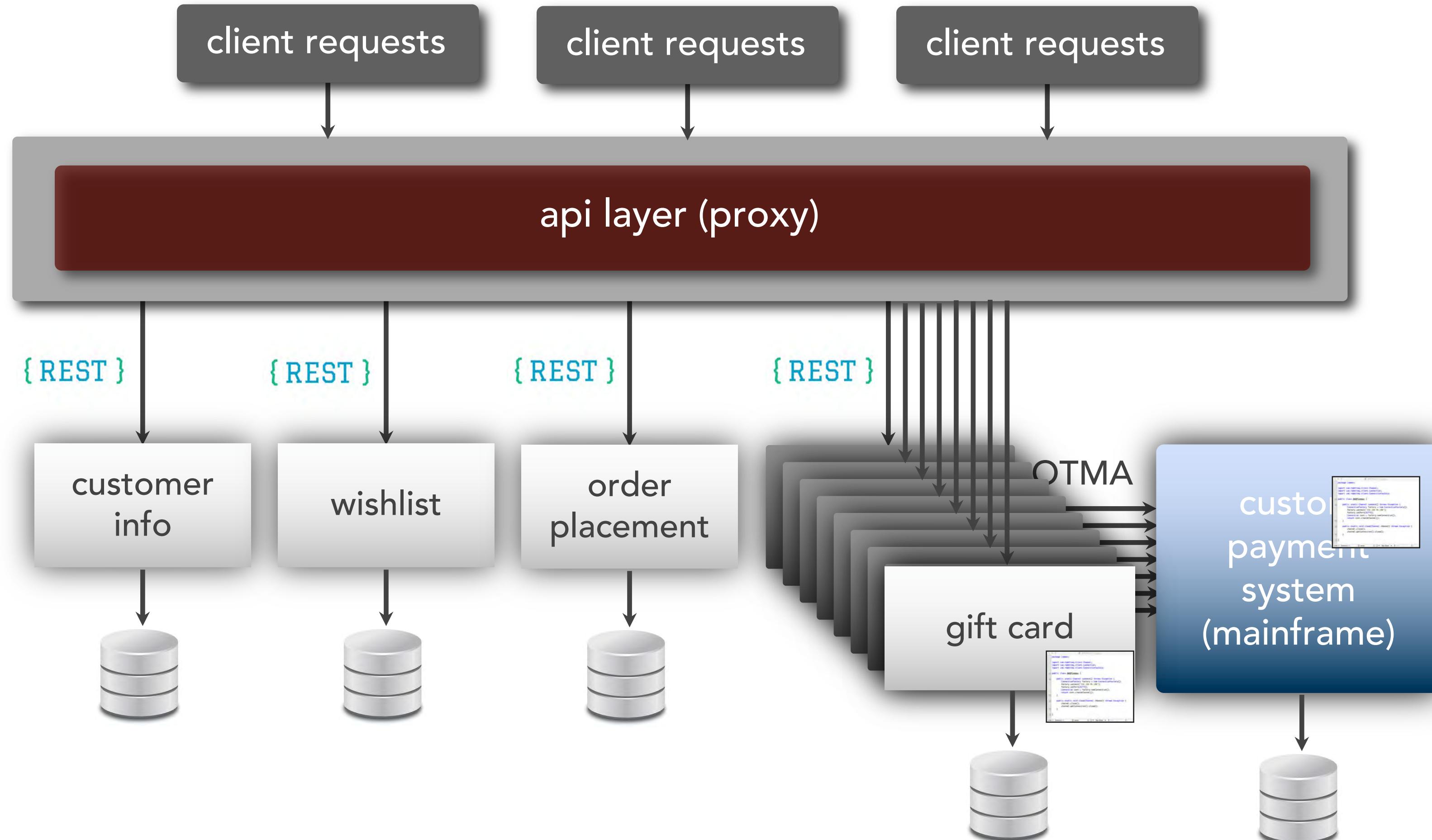
# service types

## adapter services

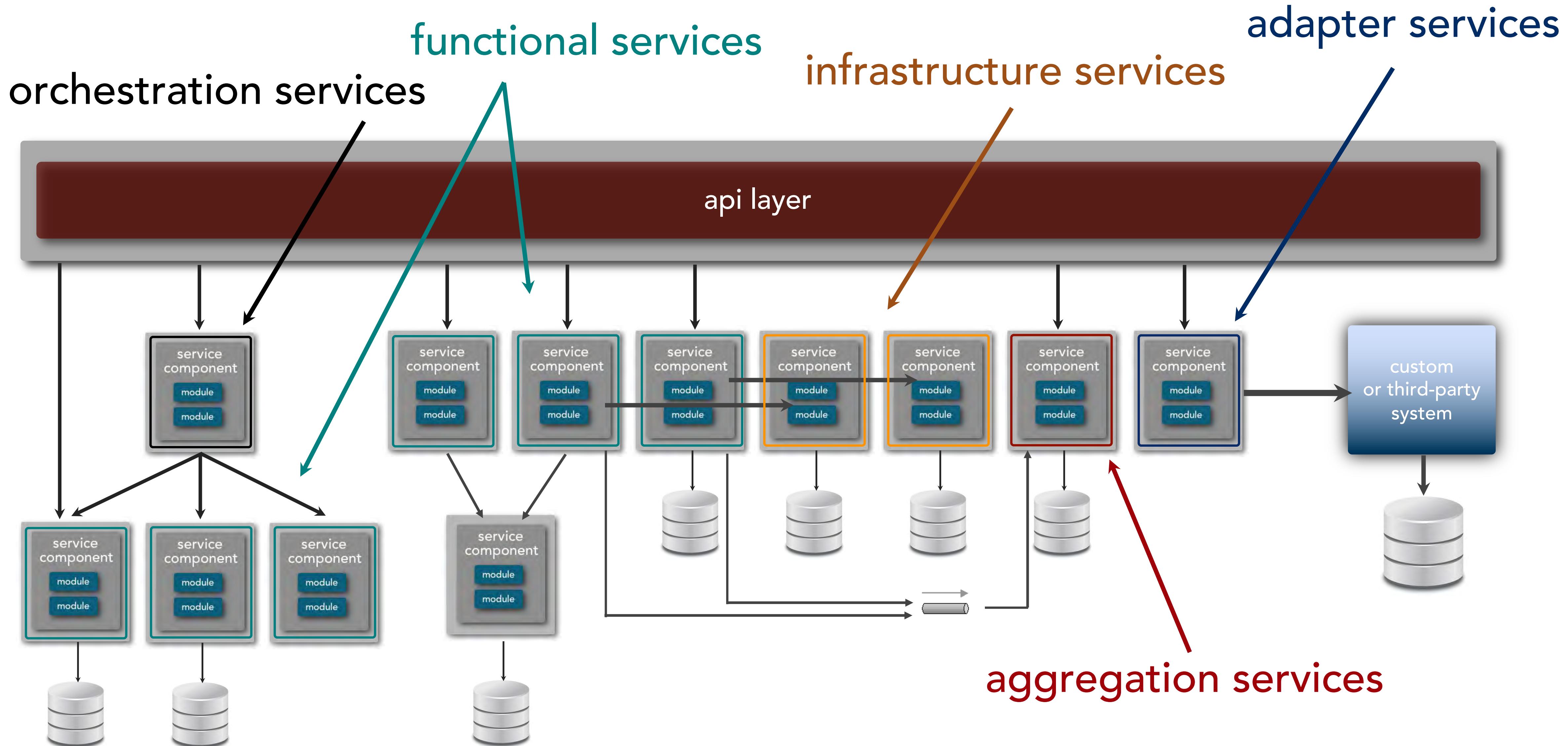


# service types

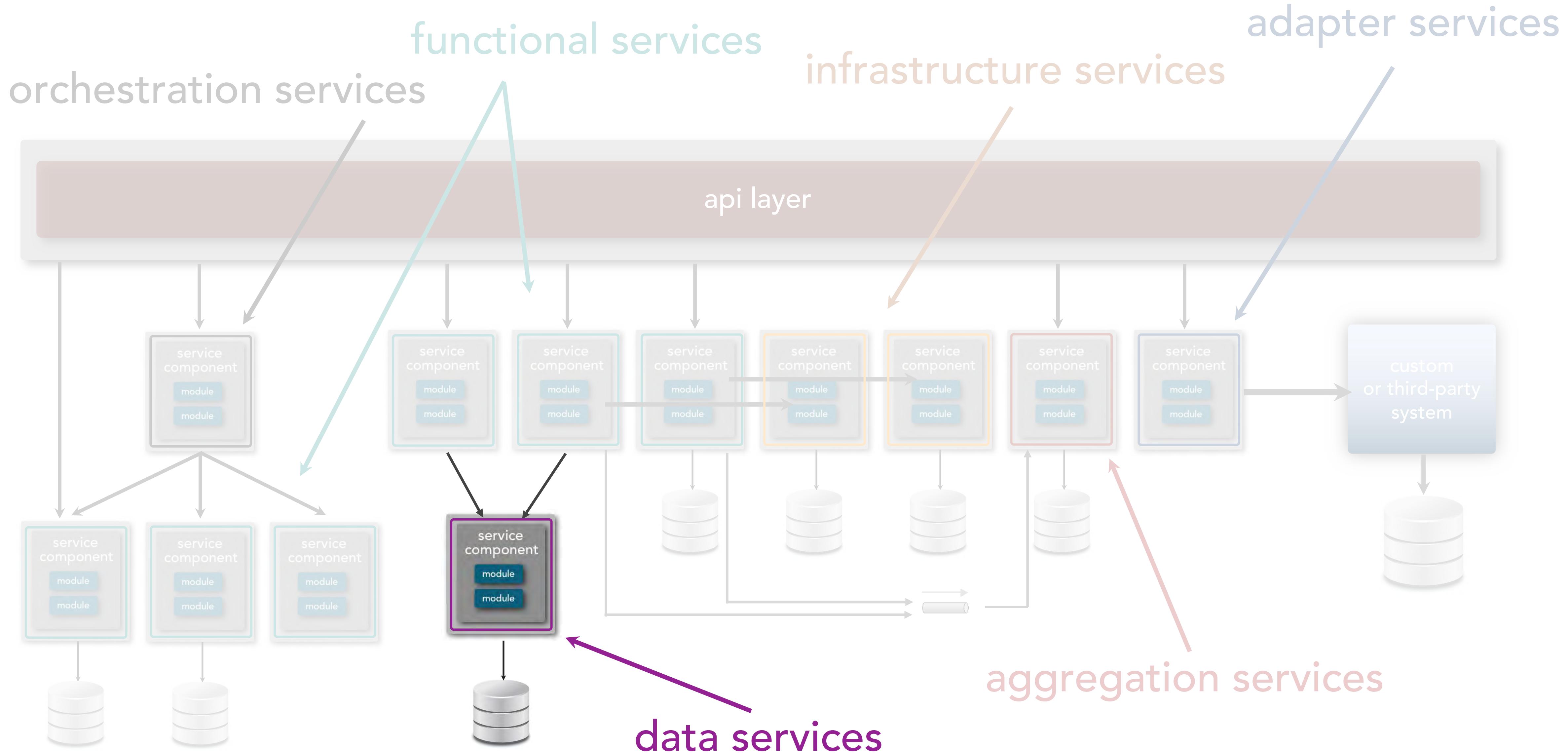
## adapter services



# service types

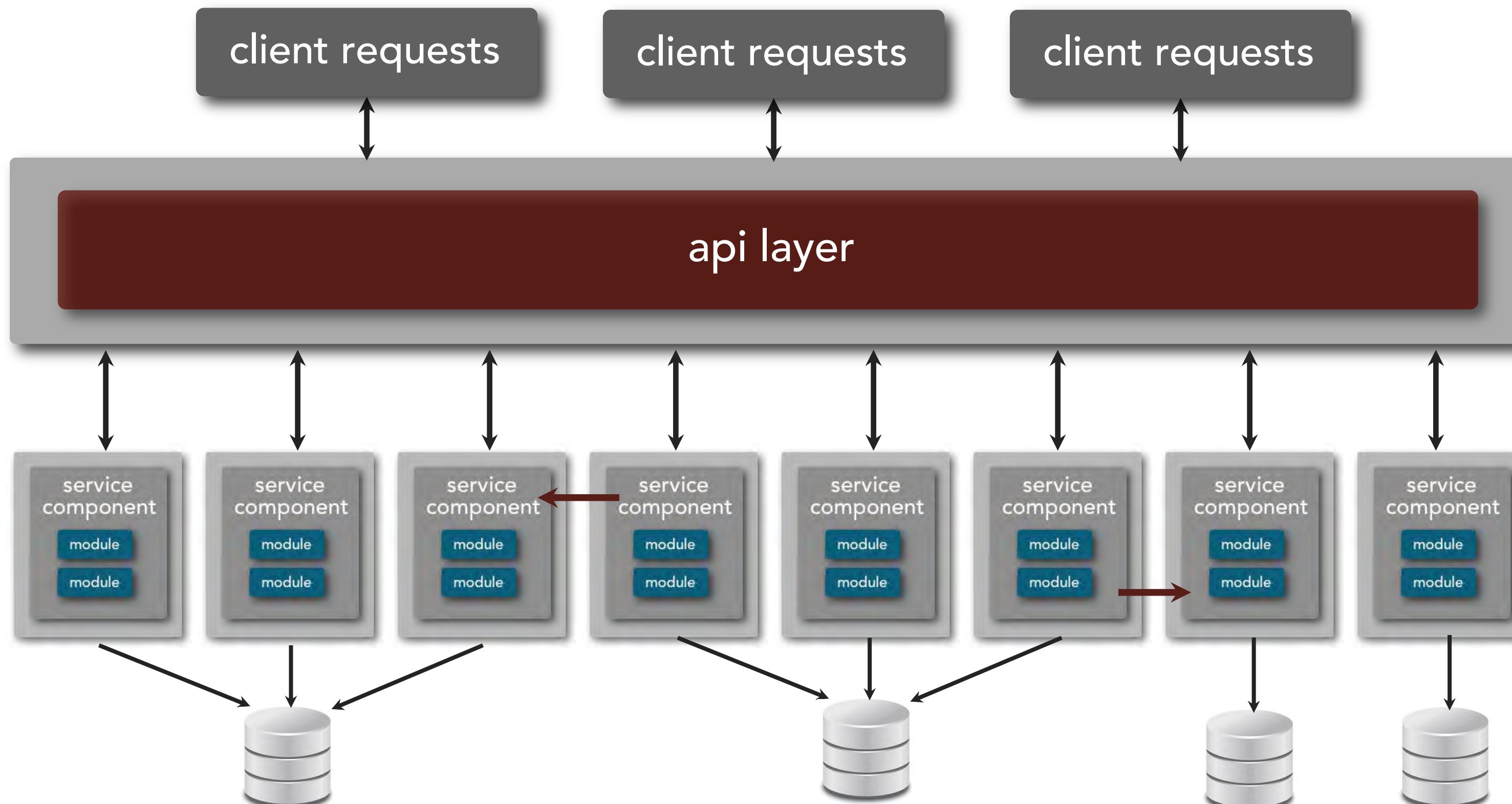


# service types



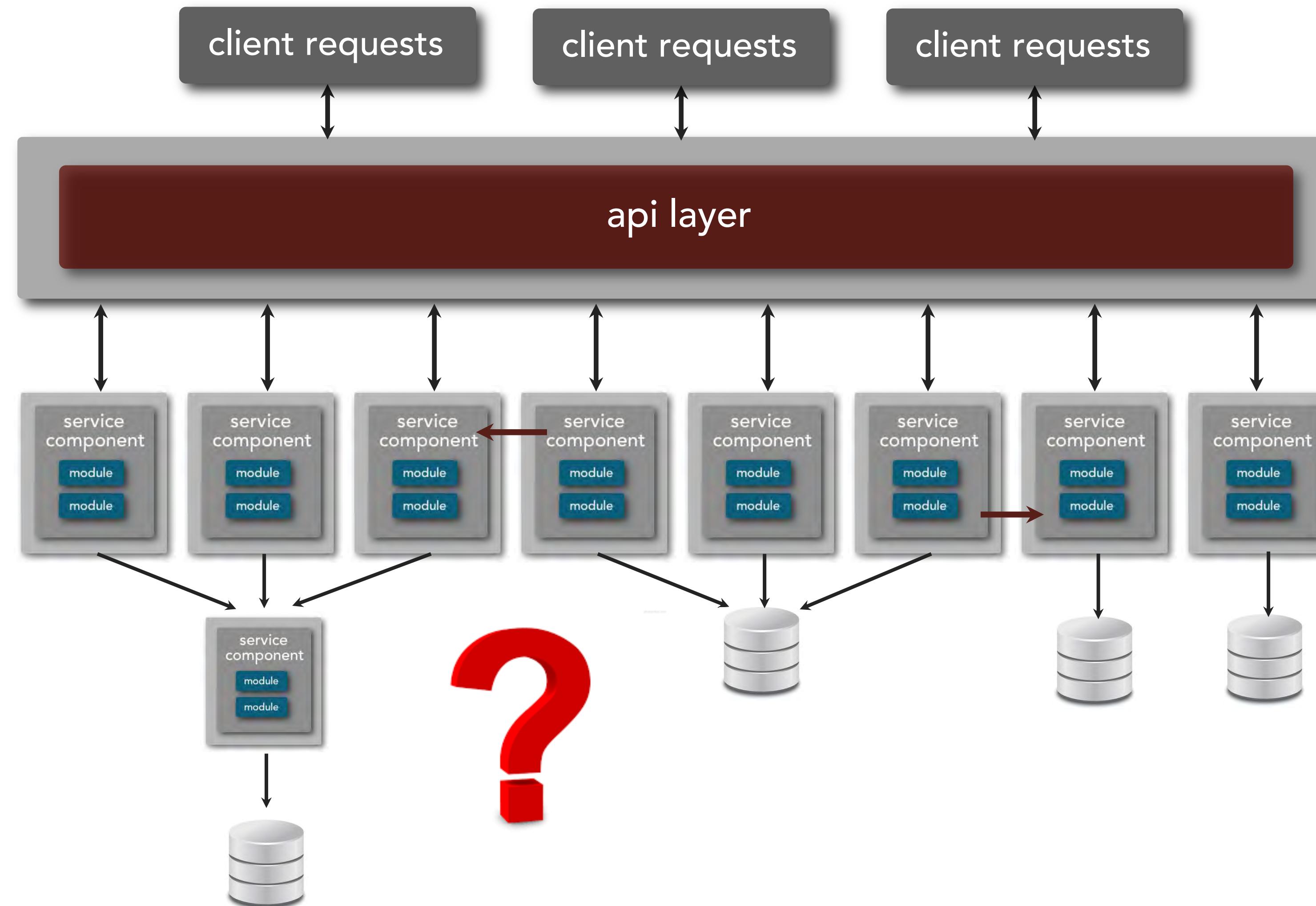
# service types

## data services



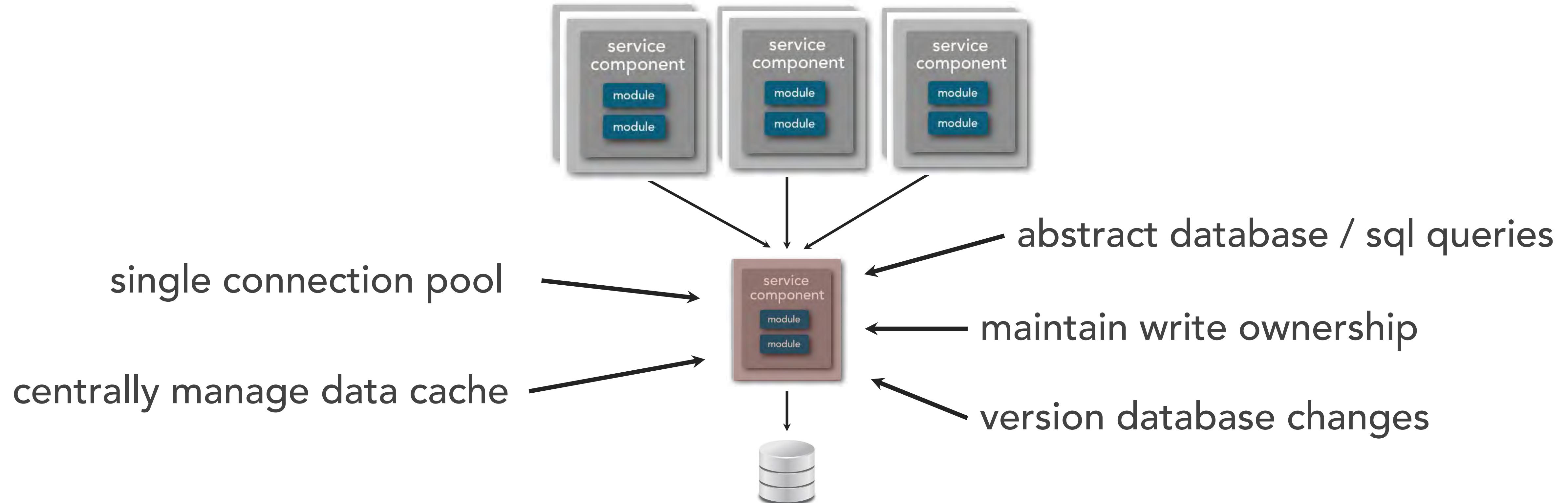
# service types

## data services



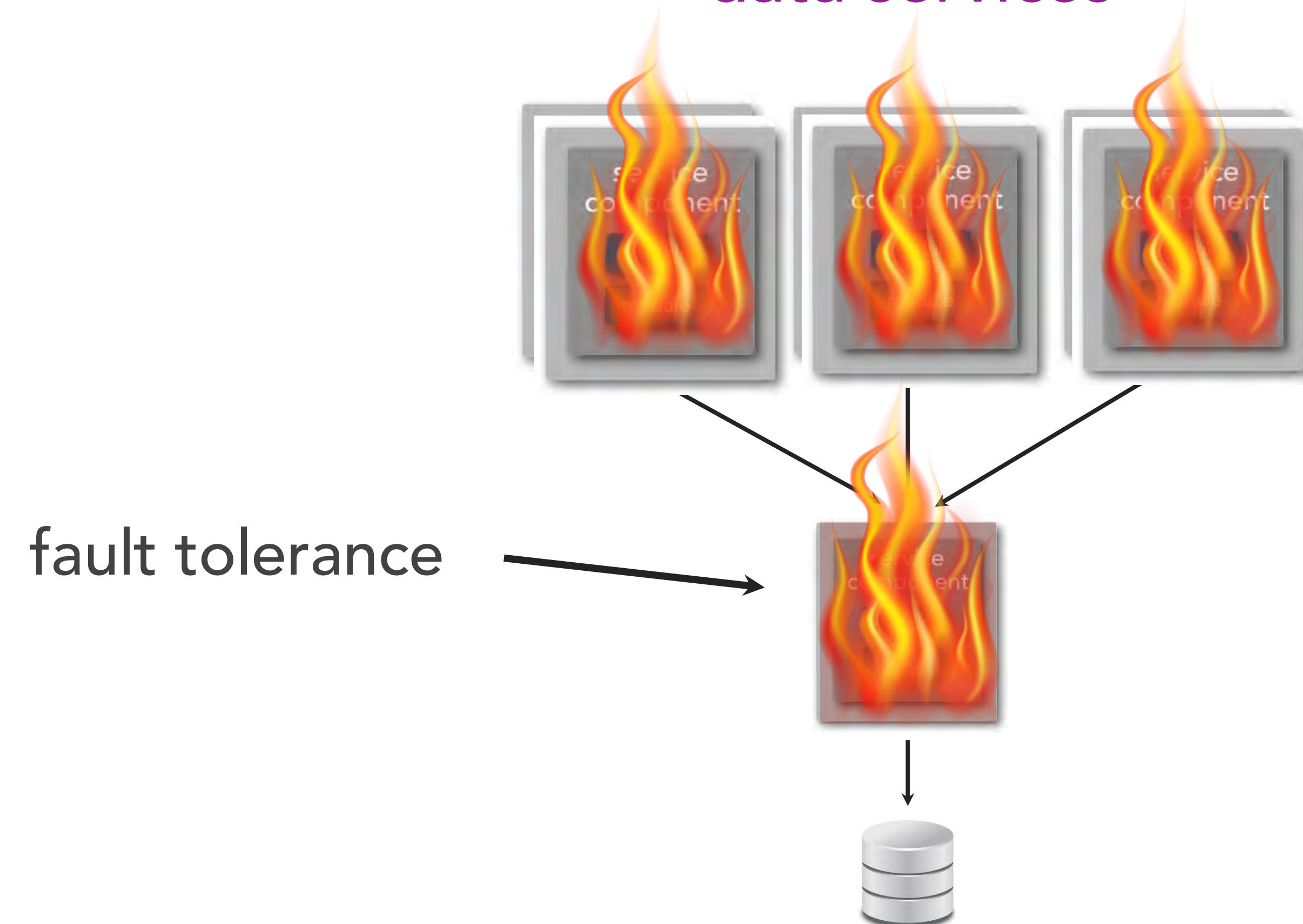
# service types

## data services



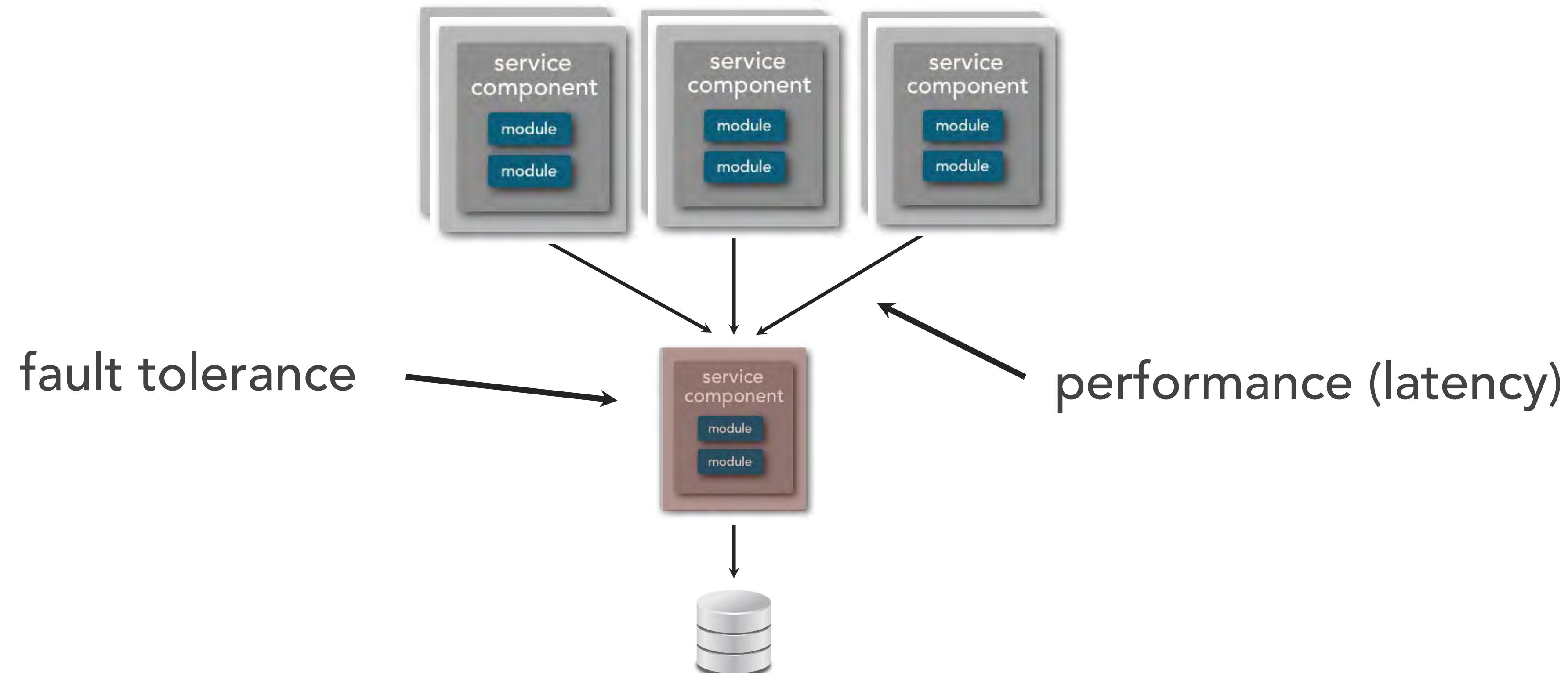
# service types

## data services



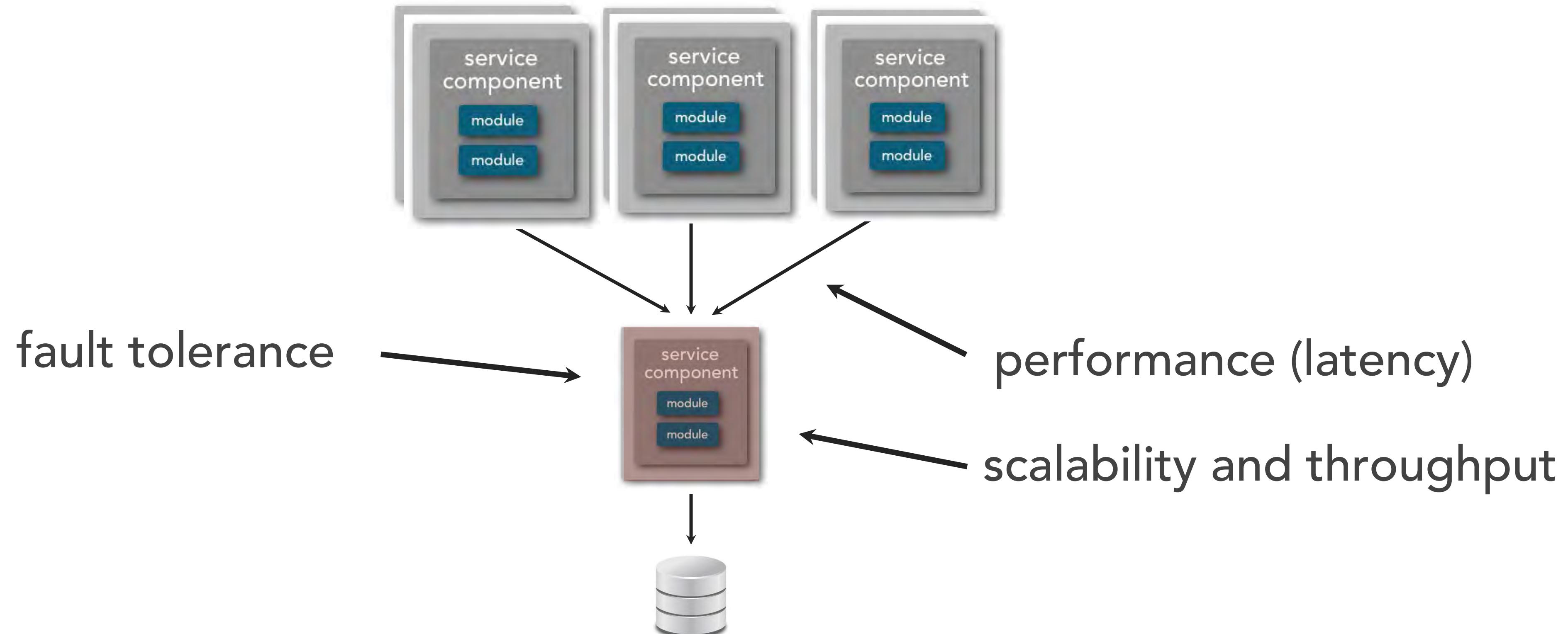
# service types

## data services



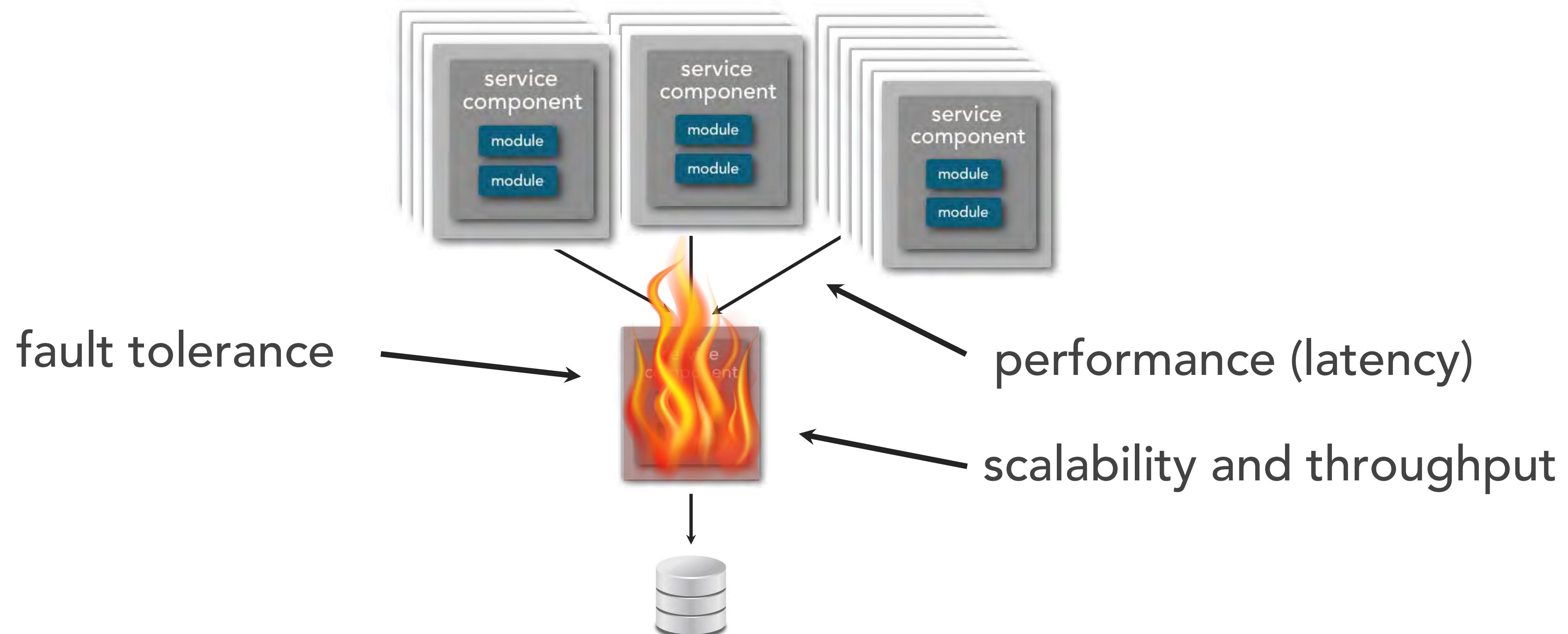
# service types

## data services



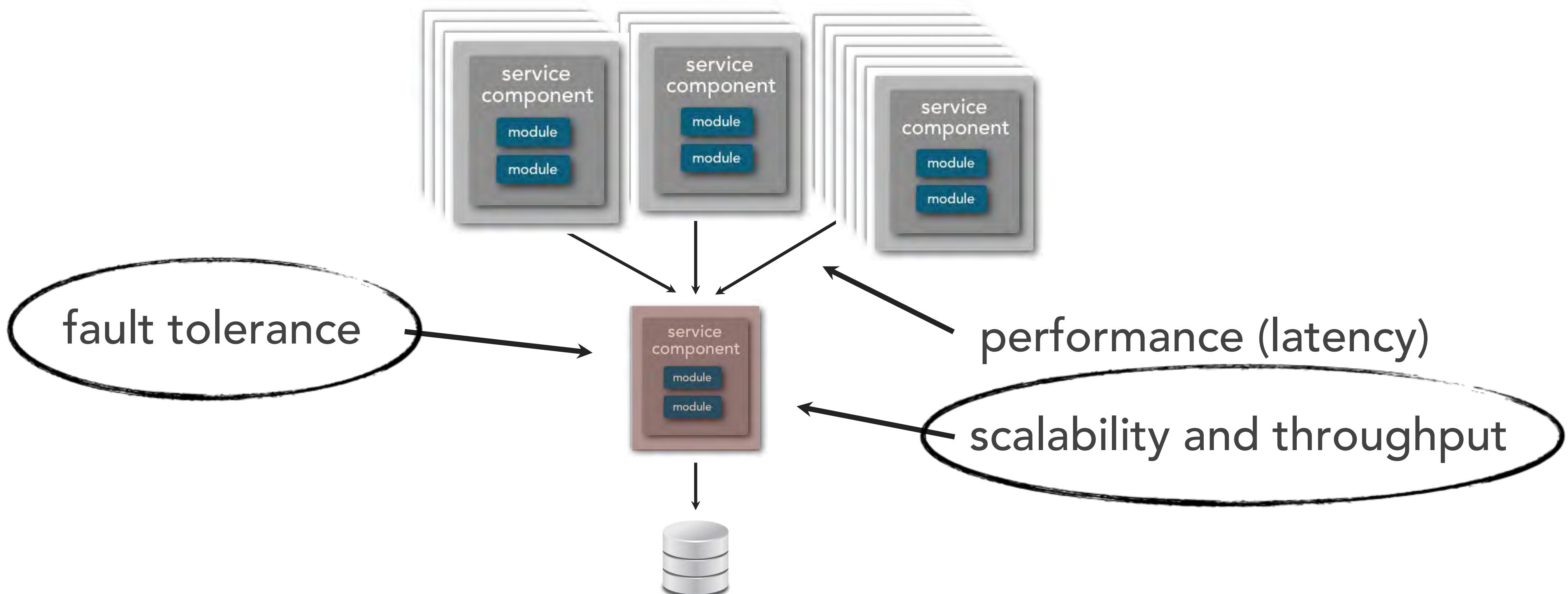
# service types

## data services



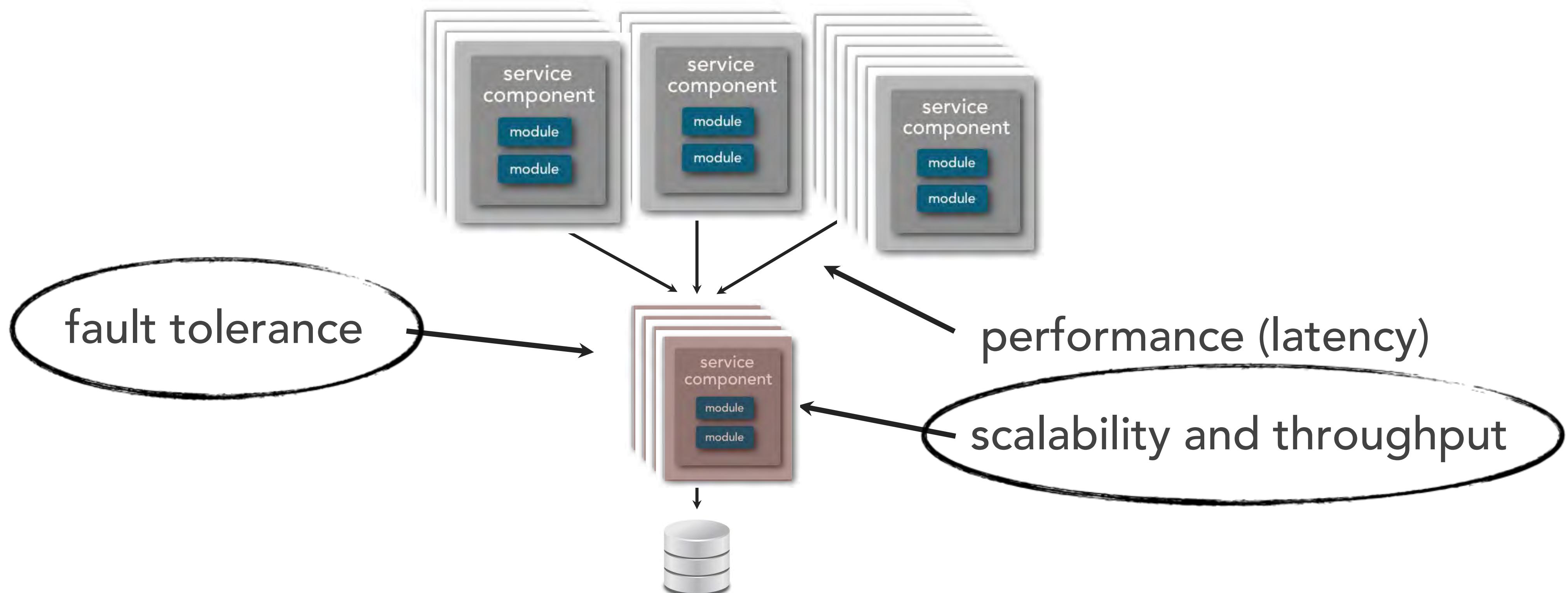
# service types

## data services



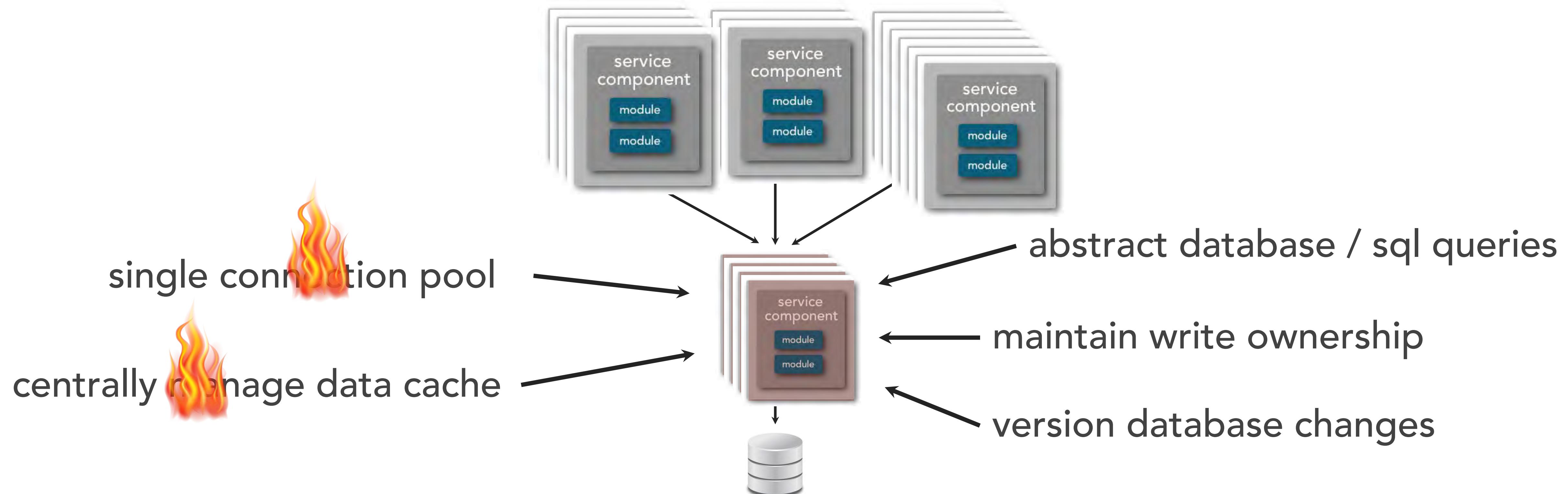
# service types

## data services



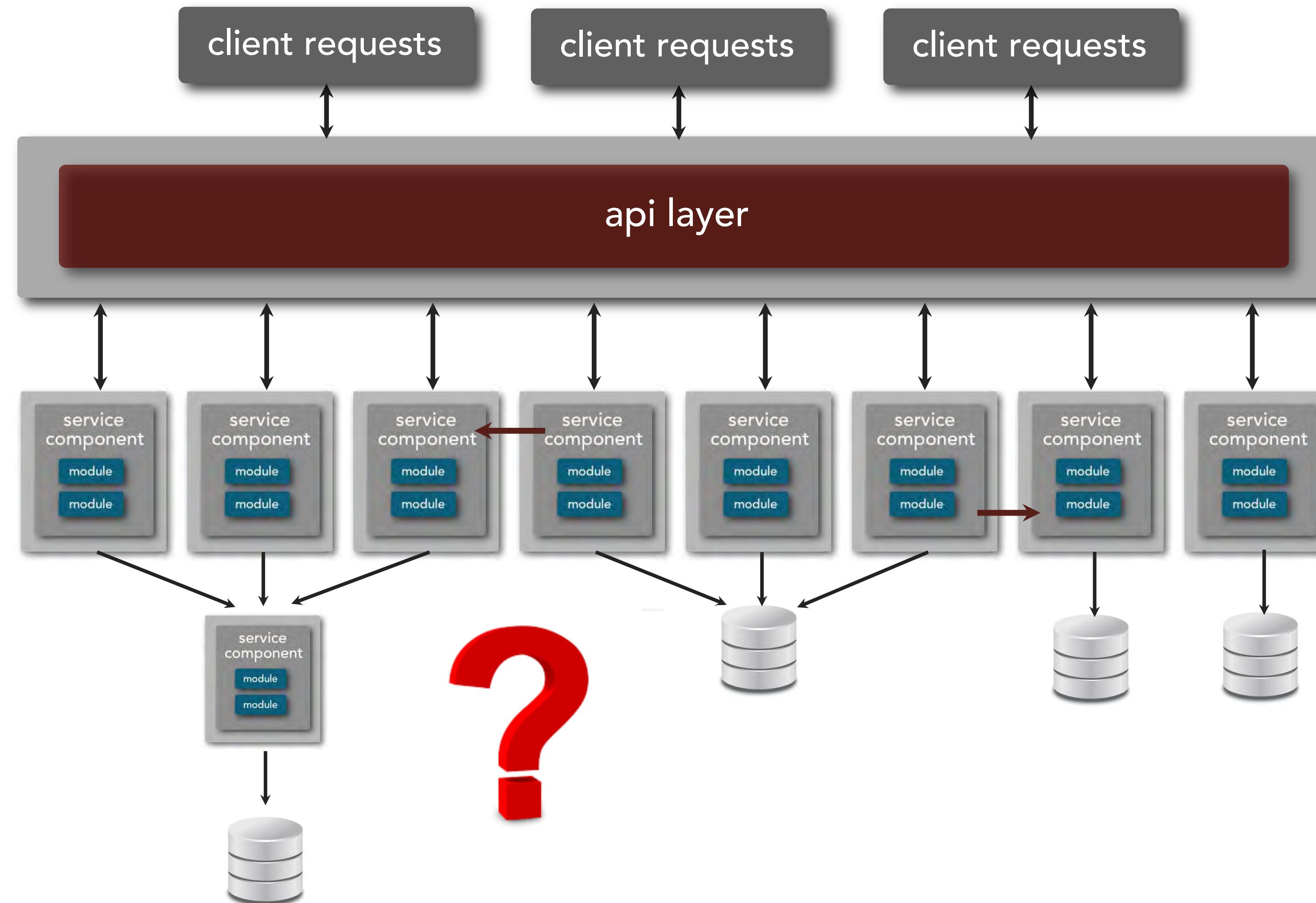
# service types

## data services

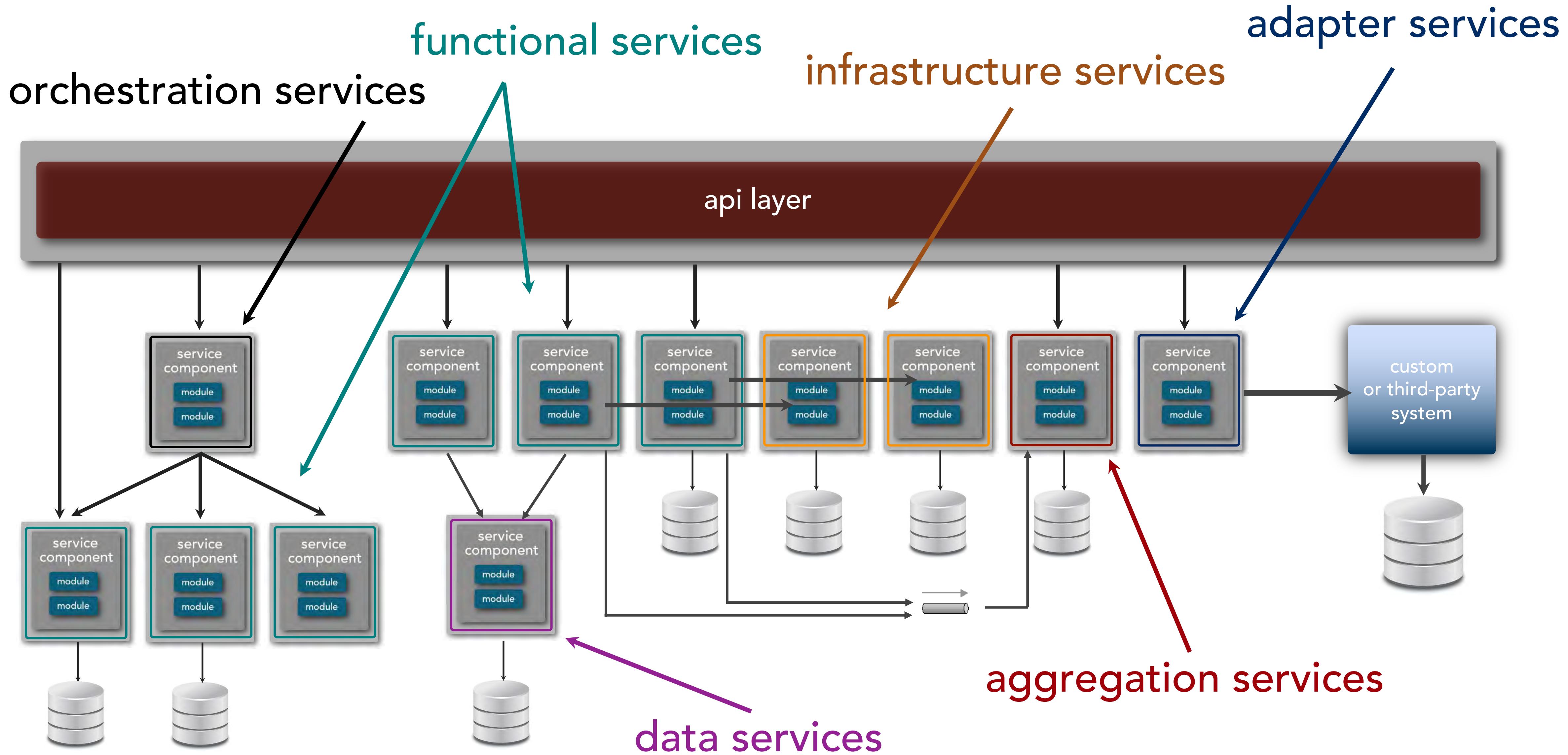


# service types

## data services



# service types



# service types

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface FunctionalService {}
```

---

```
@ServiceEntrypoint  
@FunctionalService  
public class PaymentServiceAPI {  
    ...  
}
```

# service types

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface OrchestrationService {}
```

```
@ServiceEntrypoint  
@OrchestrationService  
public class CustomerRegistration {  
    ...  
}
```

# service types

```
[System.AttributeUsage(System.AttributeTargets.Class)]  
class FunctionalService : System.Attribute {}  
  
[ServiceEntryPoint]  
[FunctionalService]  
class PaymentServiceAPI {  
    ...  
}
```

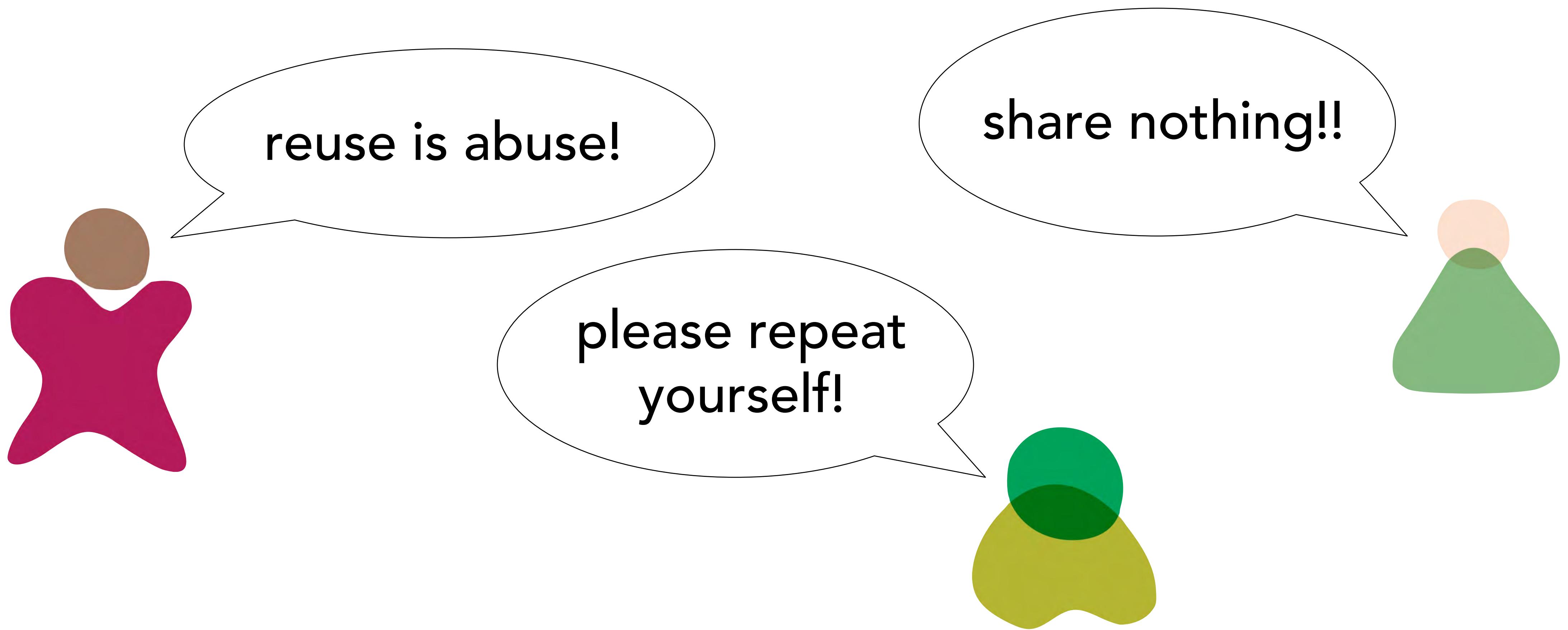
# service types

@FunctionalService	[ FunctionalService ]
@OrchestrationService	[ OrchestrationService ]
@AggregationService	[ AggregationService ]
@AdapterService	[ AdapterService ]
@InfrastructureService	[ InfrastructureService ]
@DataService	[ DataService ]

- ✓ service identification and service context (developer info)
- ✓ metrics gathering to identify structural decay
  - e.g., no more than 10% orchestration services within an application context
  - e.g., no more than 5% aggregation services within an application context
- ✓ architectural governance
  - e.g., adapter services cannot own or access data

# Code Sharing Techniques

# code sharing techniques



# code sharing techniques



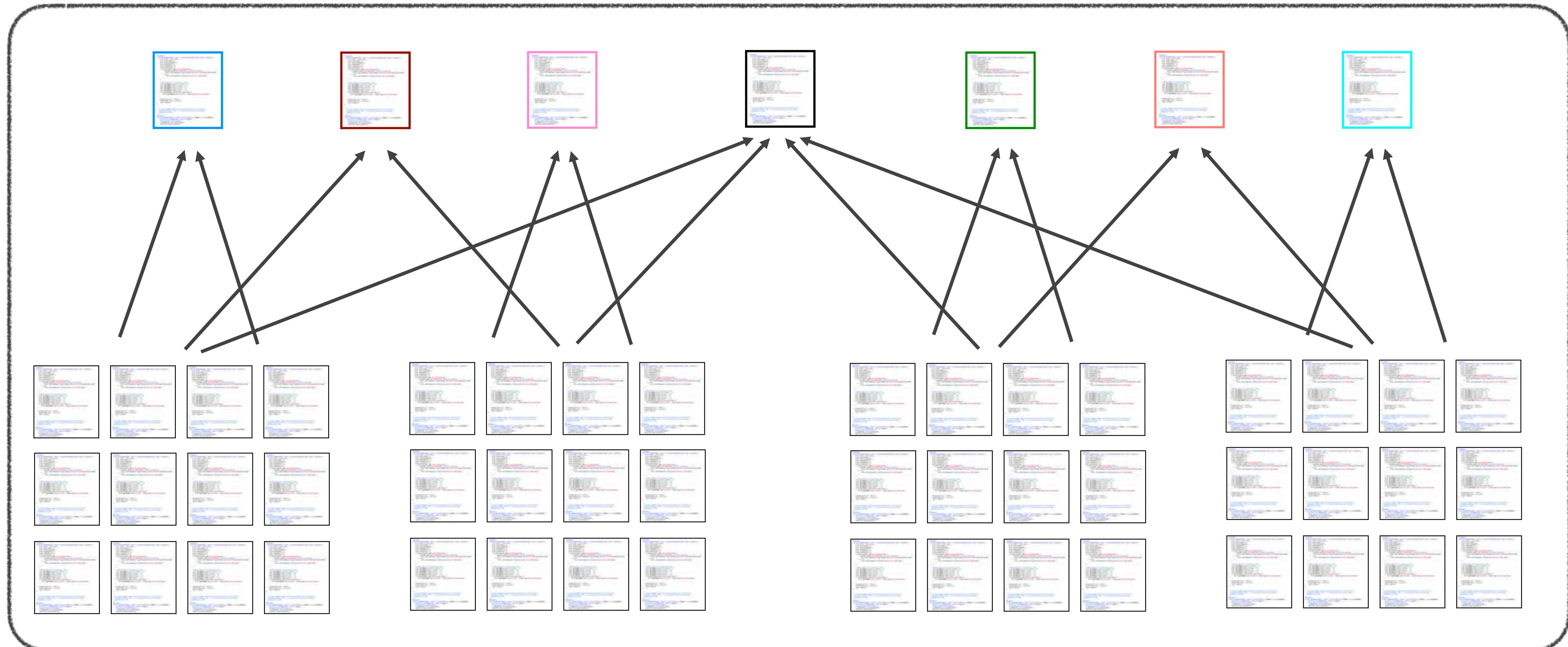
# code sharing techniques



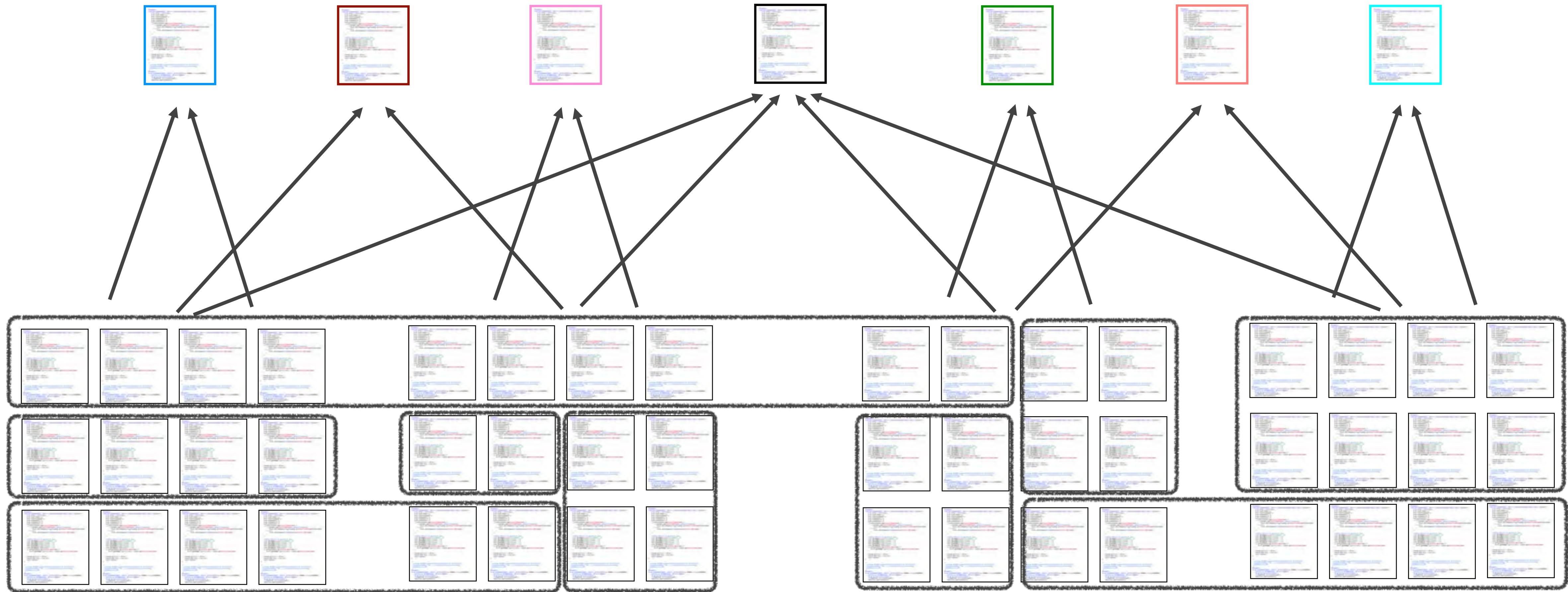
# code sharing techniques



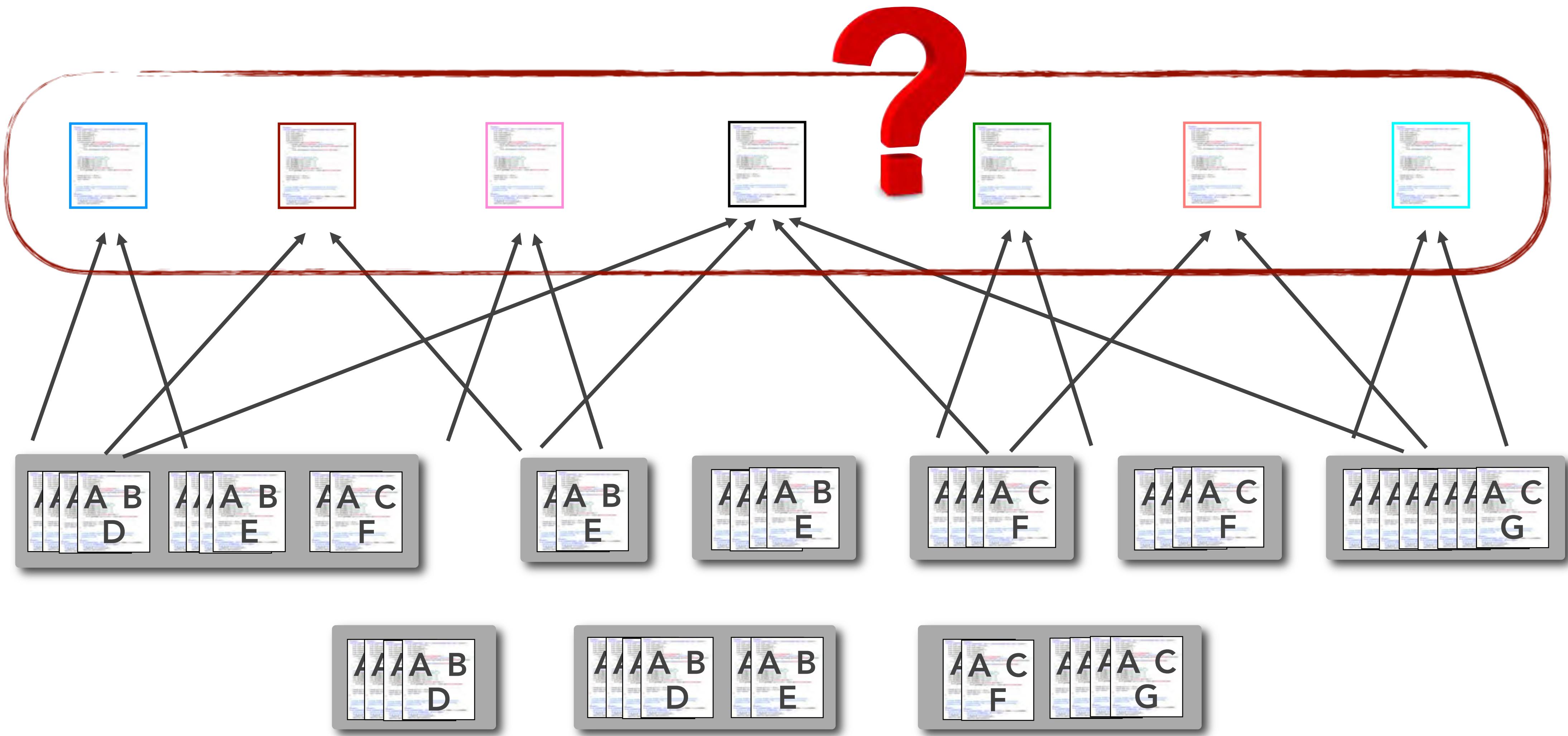
# code sharing techniques



# code sharing techniques



# code sharing techniques



# code sharing techniques

## code replication

ServiceEntrypoint.java

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface ServiceEntrypoint {}
```

```
@ServiceEntrypoint  
public class PaymentServiceAPI {  
    ...  
}
```

# code sharing techniques

## code replication

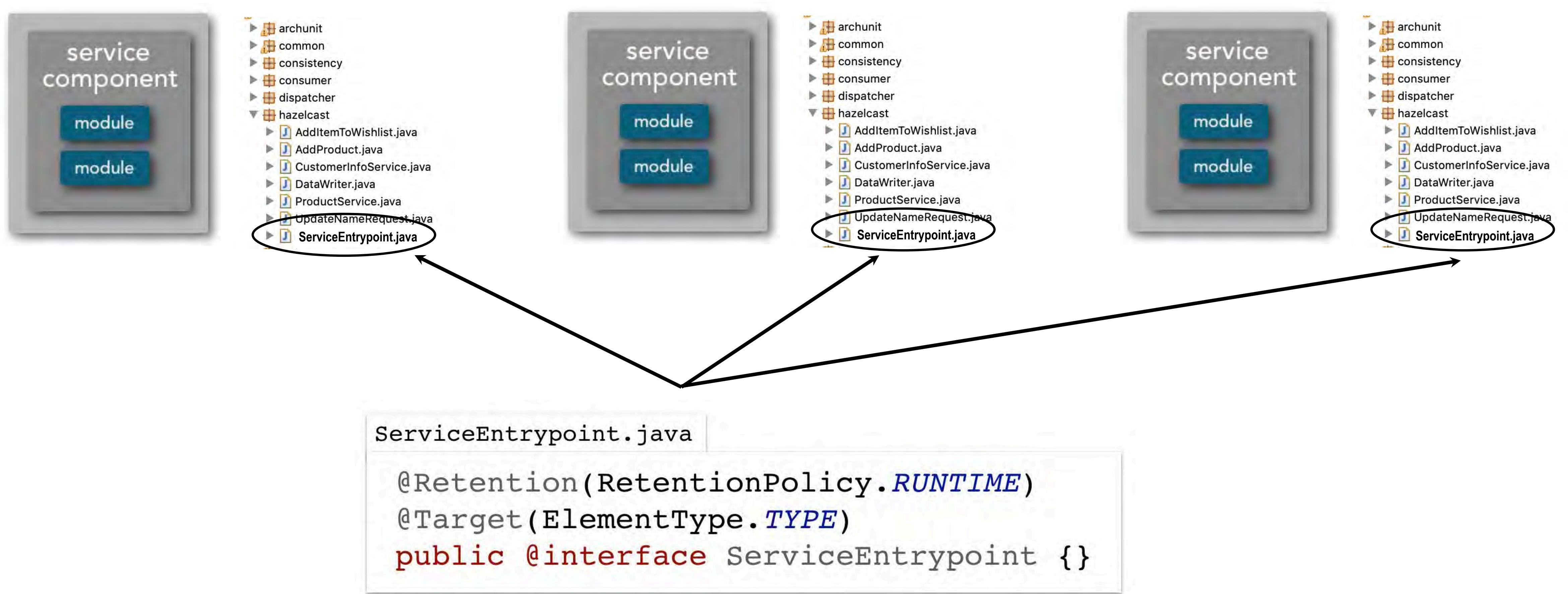
ServiceEntrypoint.cs

```
[AttributeUsage(AttributeTargets.Class)]
class ServiceEntrypoint : Attribute {}
```

```
[ServiceEntrypoint]
class PaymentServiceAPI {
    ...
}
```

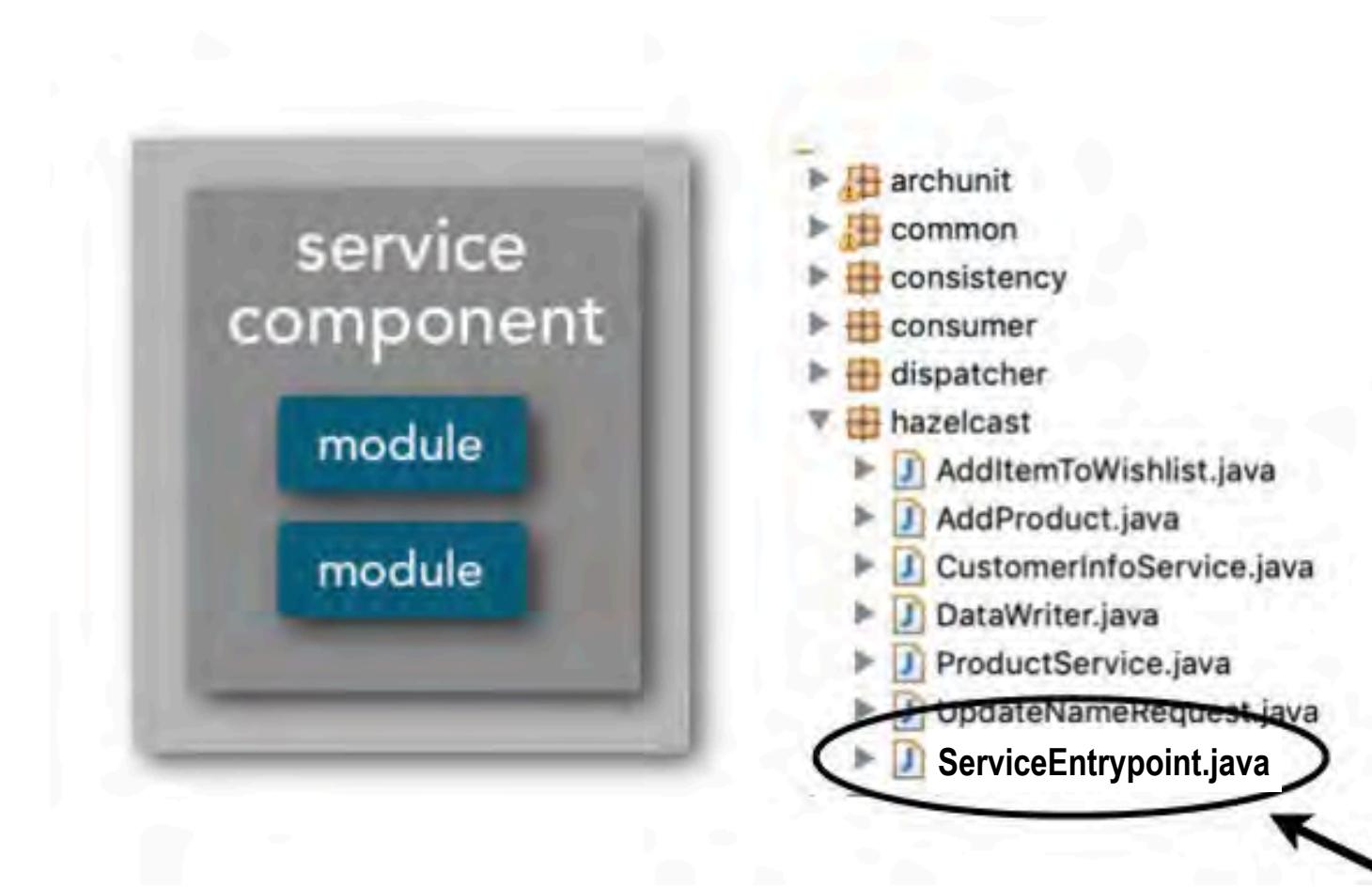
# code sharing techniques

## code replication



# code sharing techniques

## code replication



- ✓ no code sharing
- ✓ good for static code



- ✗ code changes
- ✗ difficult to expand context

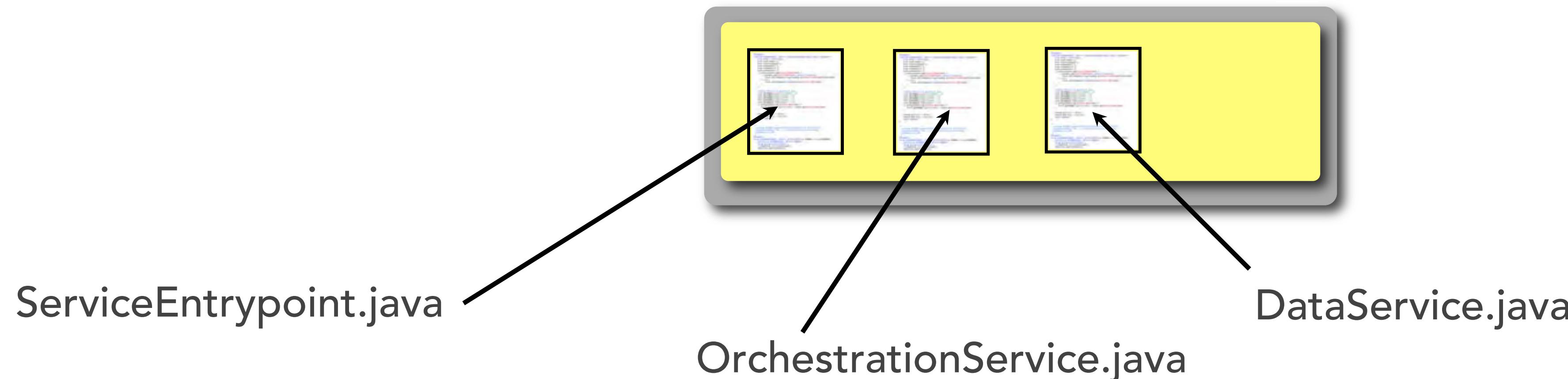
# code sharing techniques

## shared library

ServiceEntrypoint.java

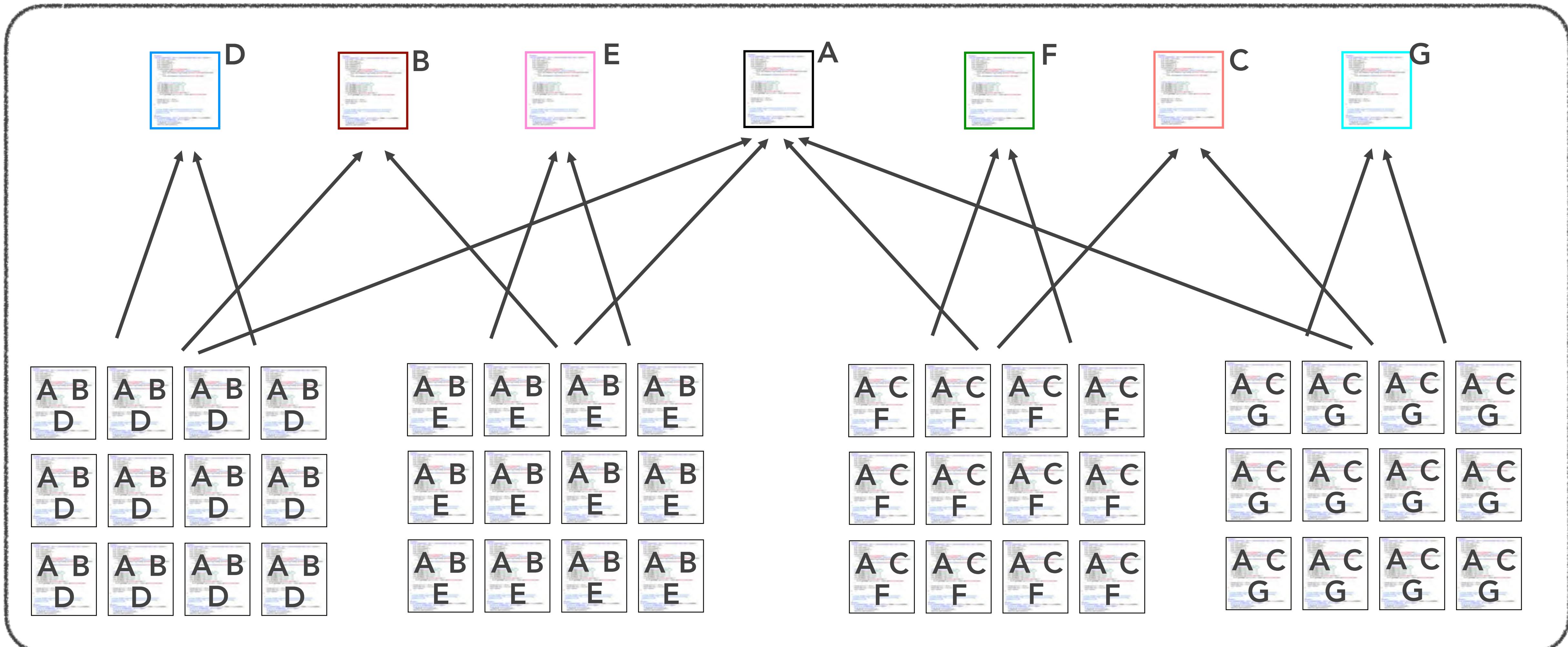
```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface ServiceEntrypoint {}
```

annotations.jar v1.0 v1.1 v1.2



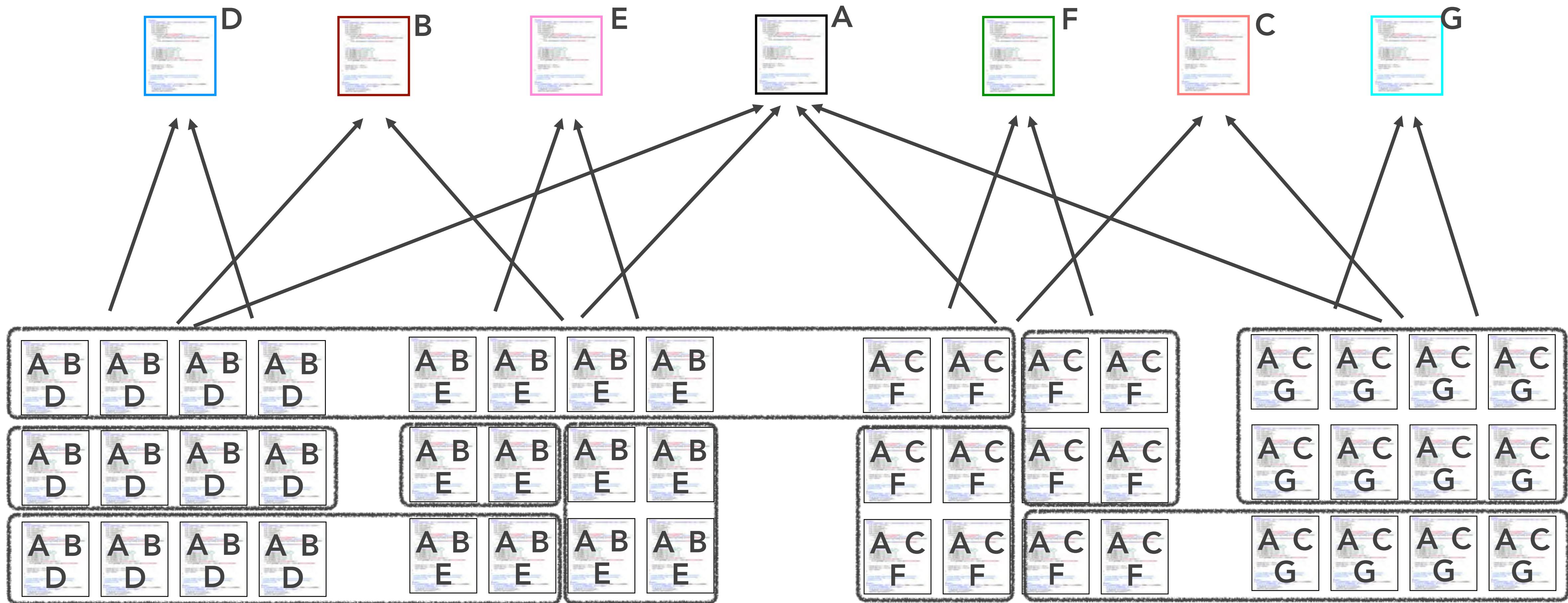
# code sharing techniques

## shared library



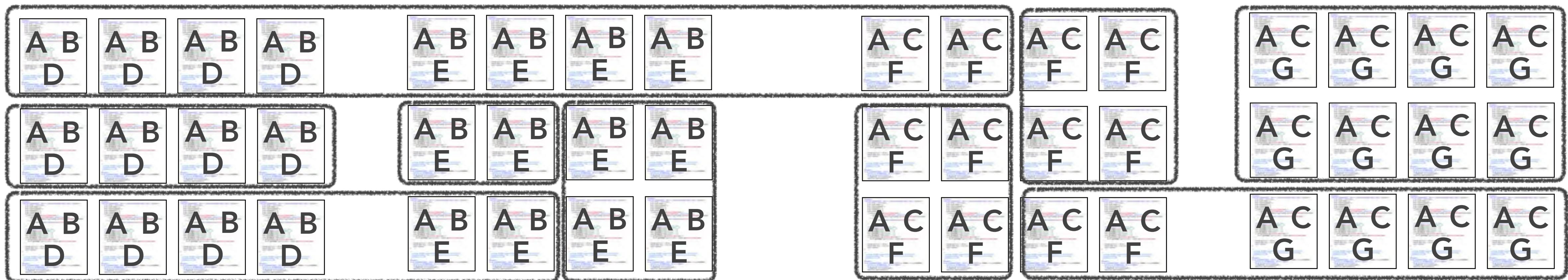
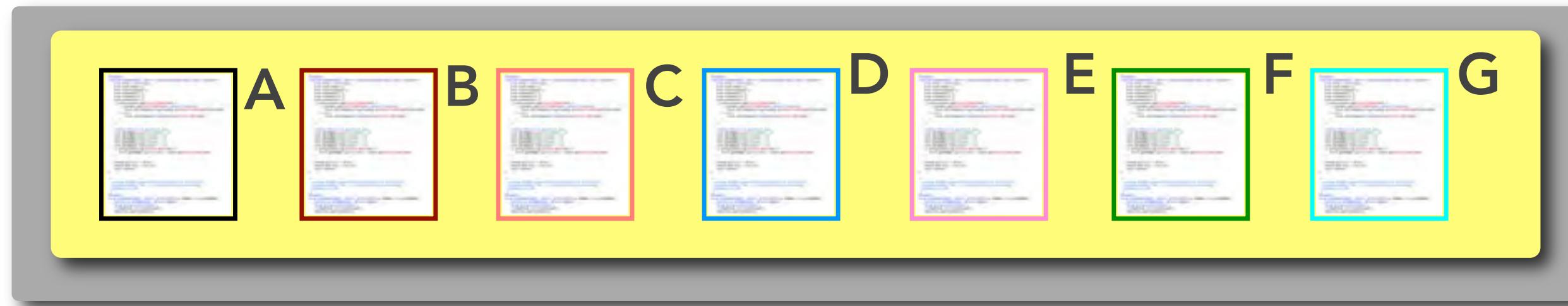
# code sharing techniques

## shared library



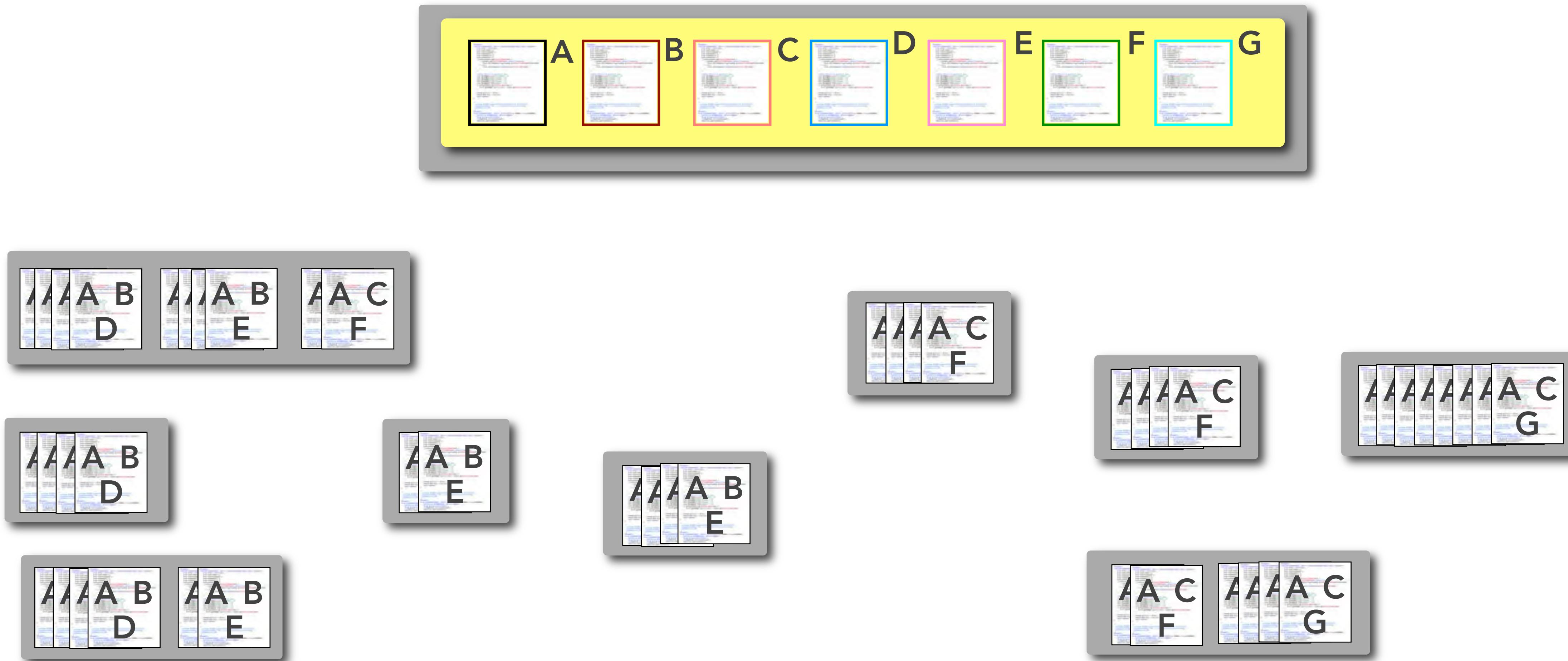
# code sharing techniques

## shared library



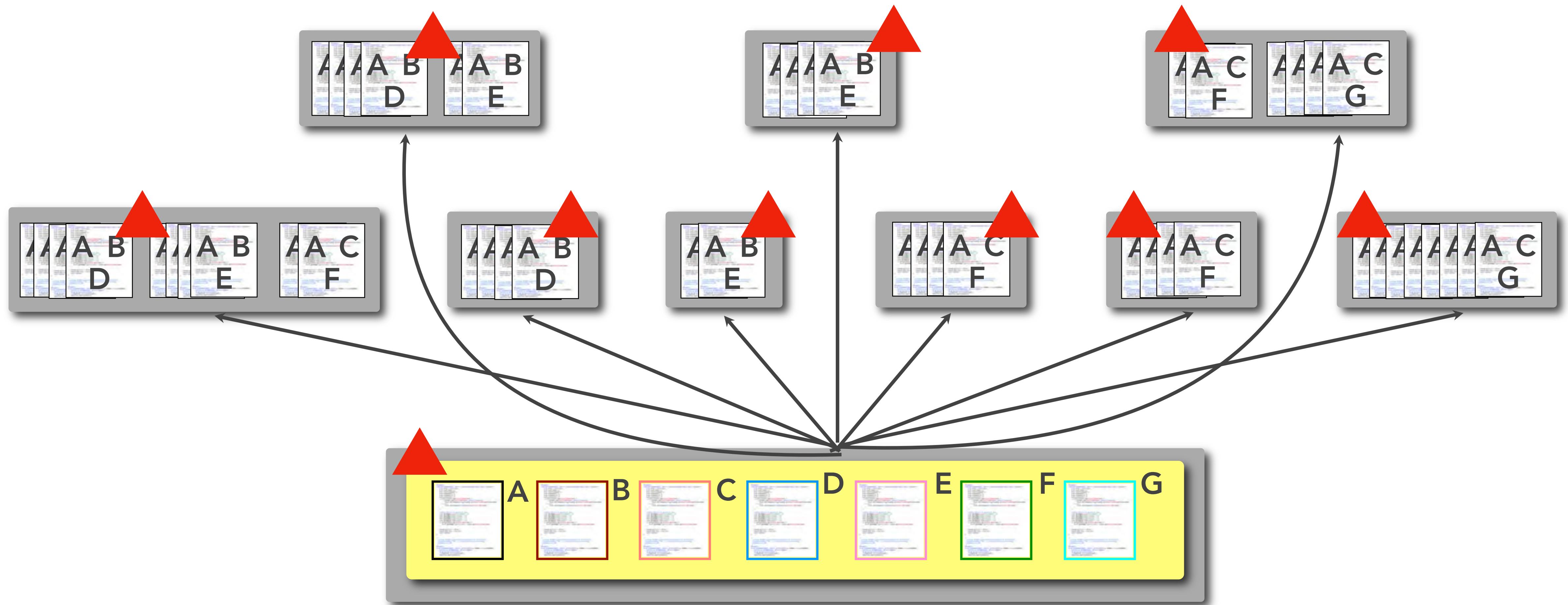
# code sharing techniques

## shared library



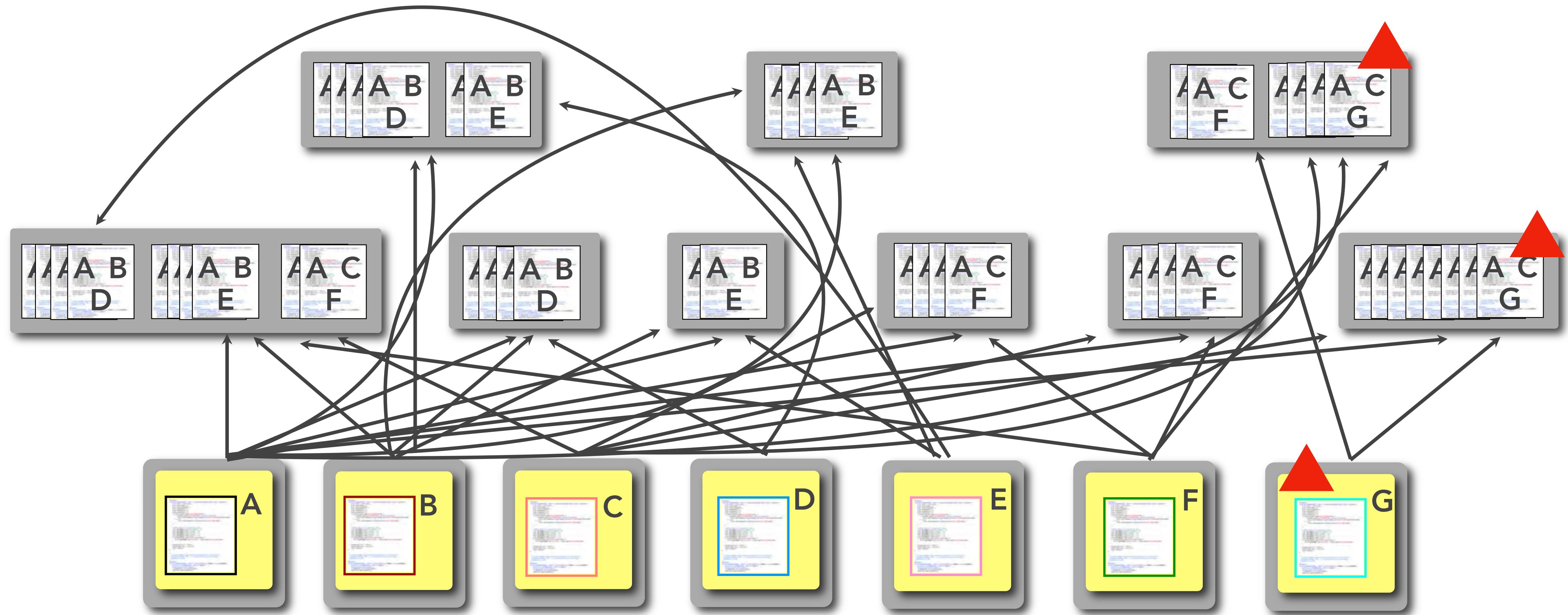
# code sharing techniques

## shared library



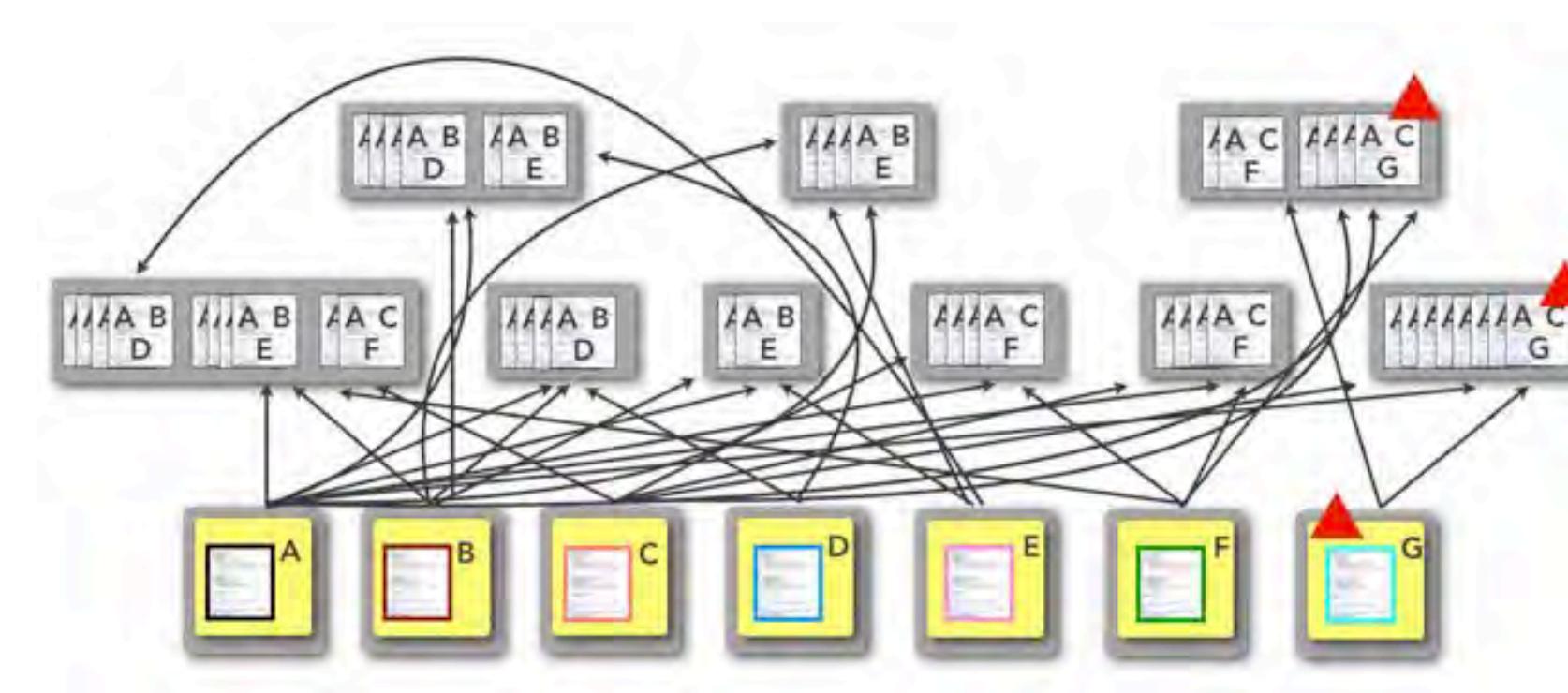
# code sharing techniques

## shared library



# code sharing techniques

## shared library



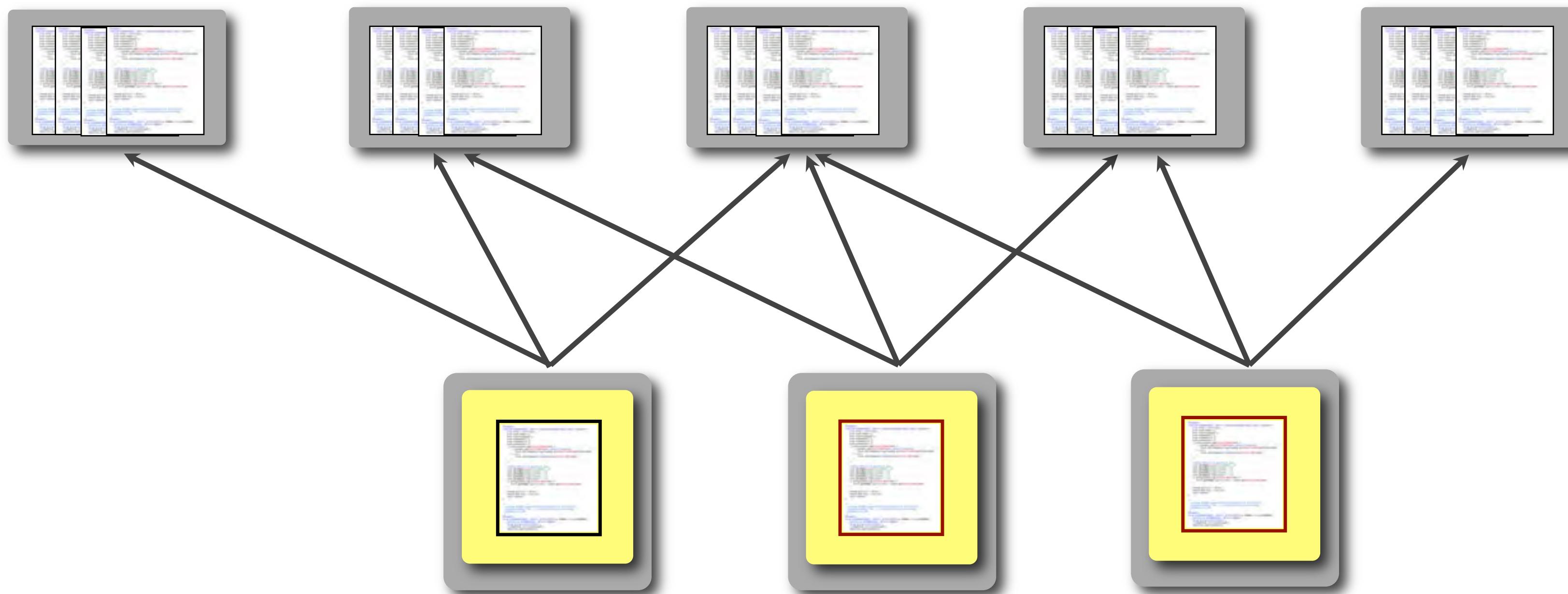
- ✓ ability to version changes
- ✓ easy to expand



- ✗ dependency management
- ✗ heterogeneous code bases

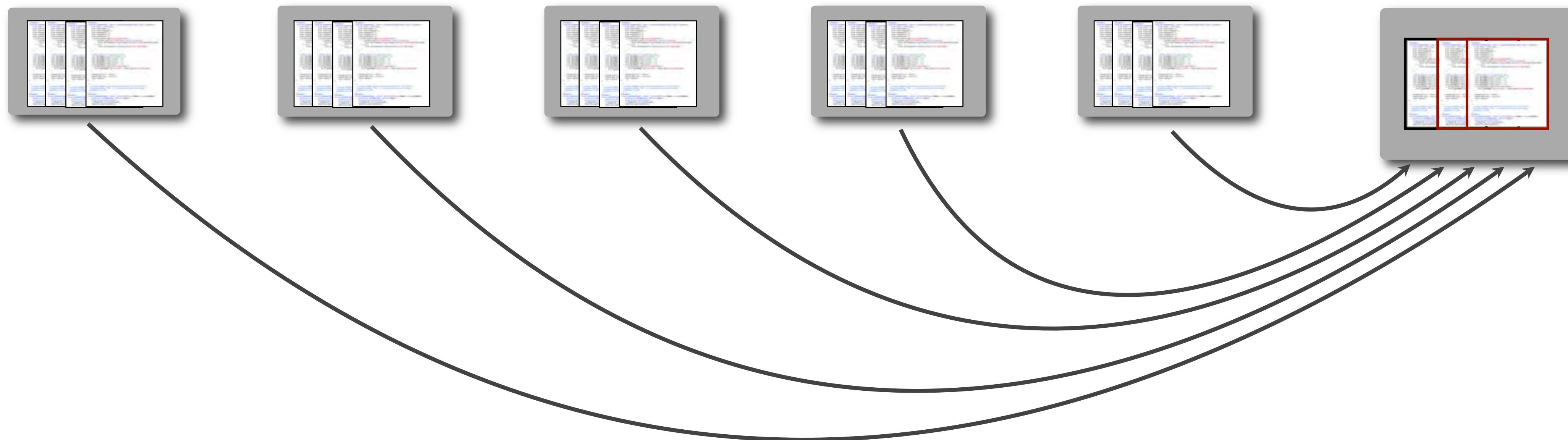
# code sharing techniques

## shared service



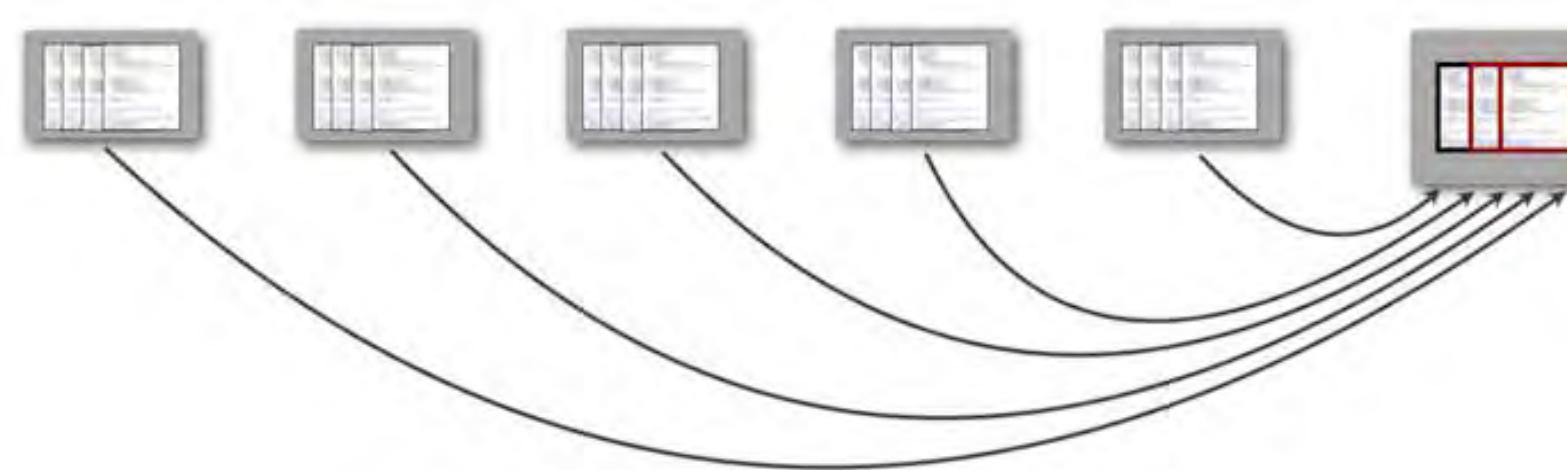
# code sharing techniques

## shared service



# code sharing techniques

## shared service



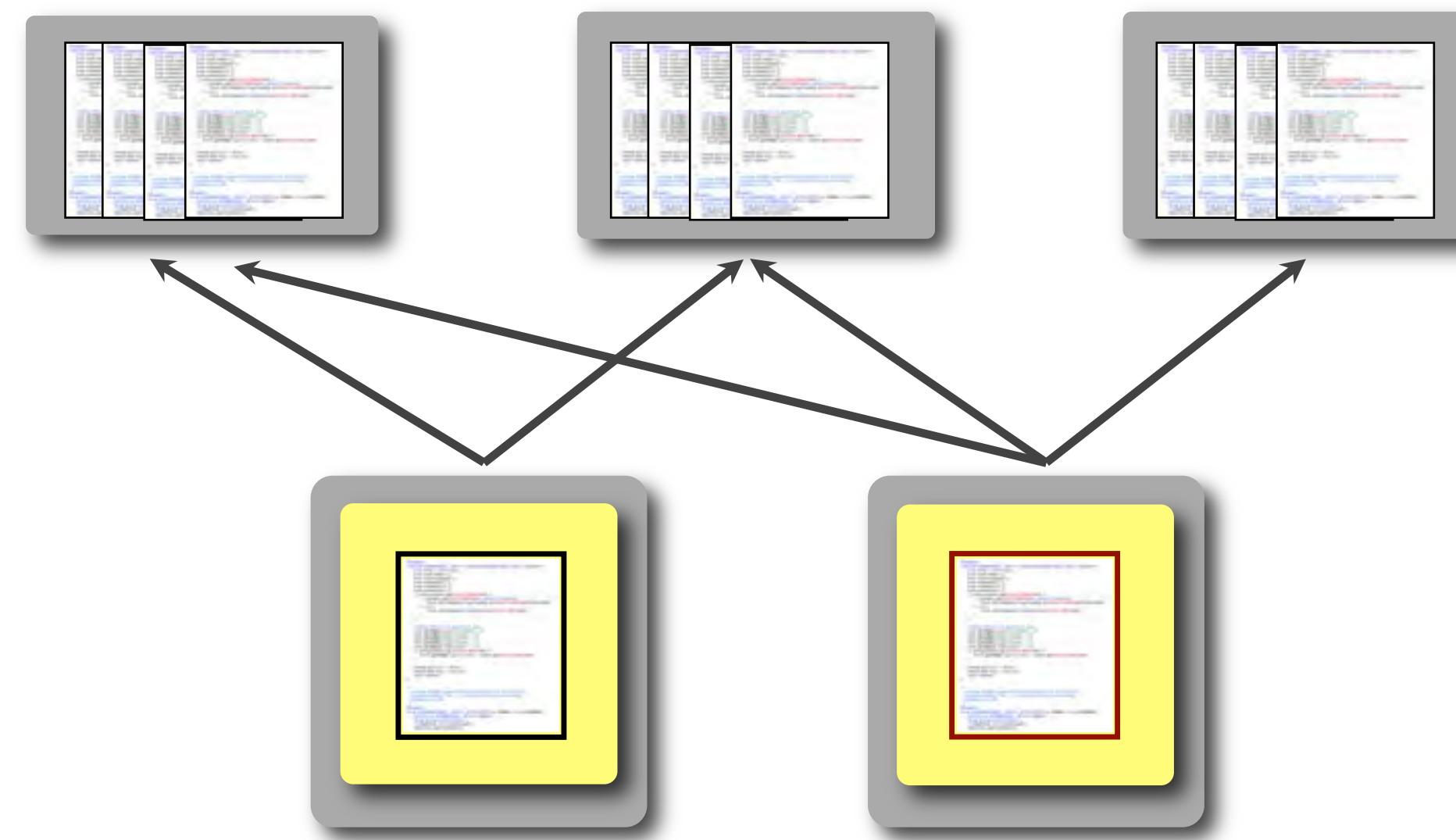
- ✓ high code volatility
- ✓ heterogeneous code



- ✗ versioning is difficult
- ✗ performance issues due to latency
- ✗ availability / fault tolerance
- ✗ scalability / throughput
- ✗ runtime changes (things break)

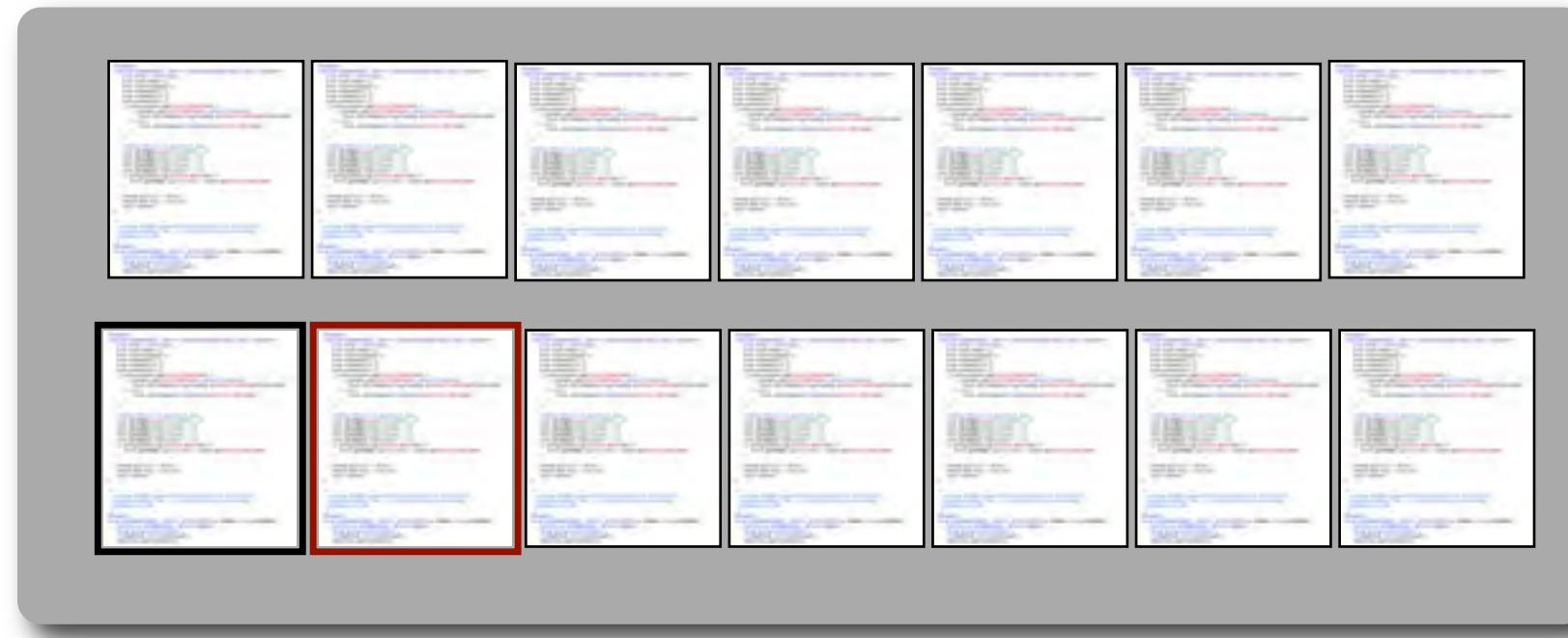
# code sharing techniques

## service consolidation



# code sharing techniques

## service consolidation

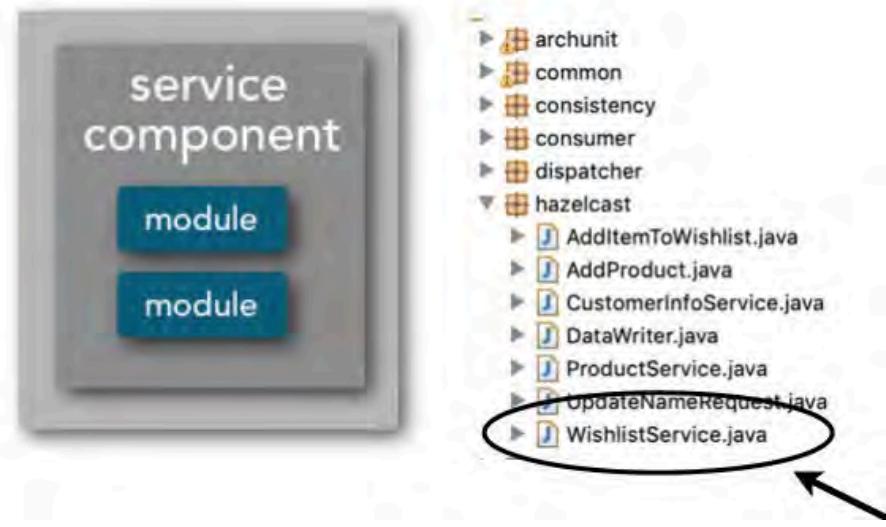


✓ no code sharing

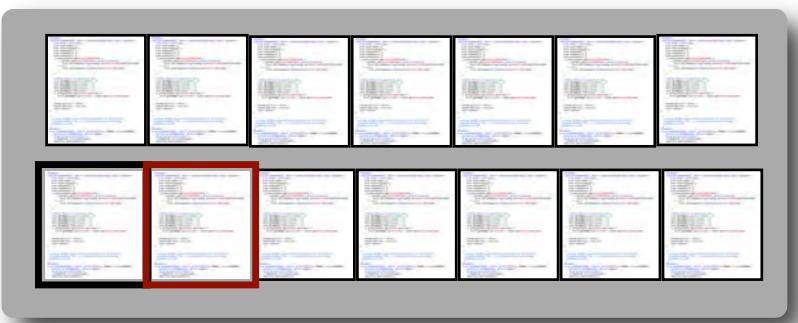


- ✗ larger testing scope
- ✗ greater risk of deployment
- ✗ less agility

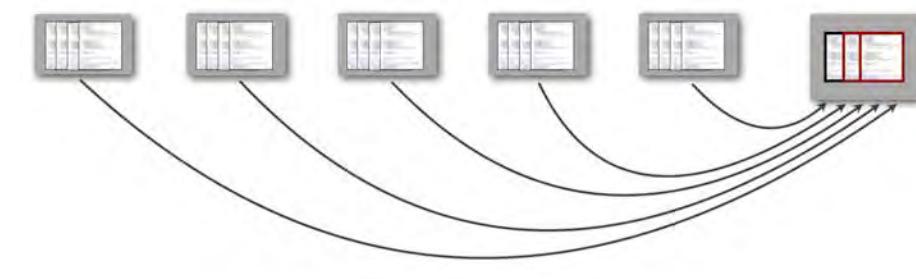
# code sharing techniques



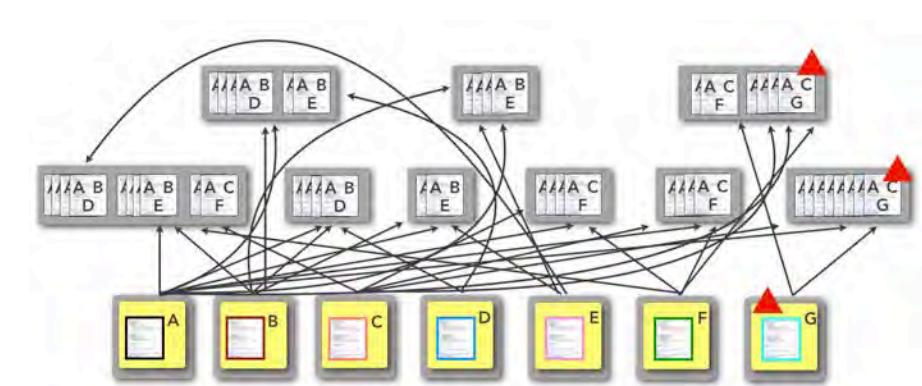
replicated  
code



consolidate  
services



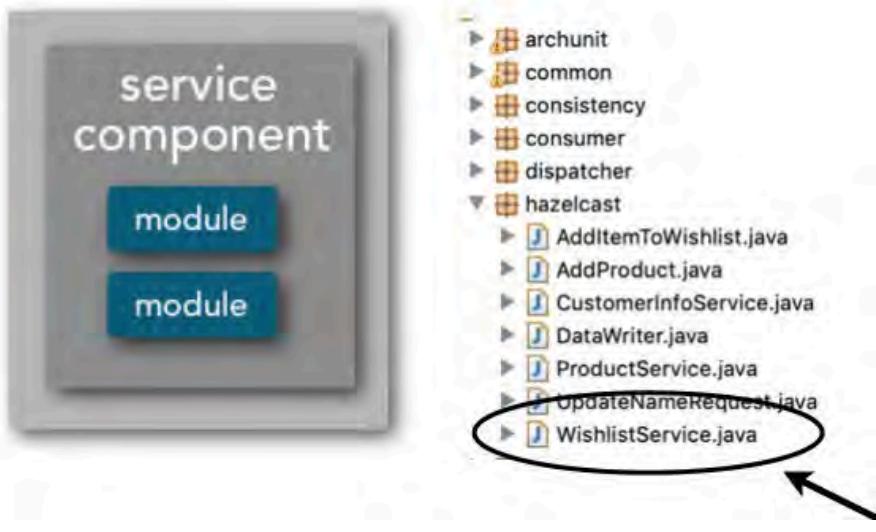
shared  
service



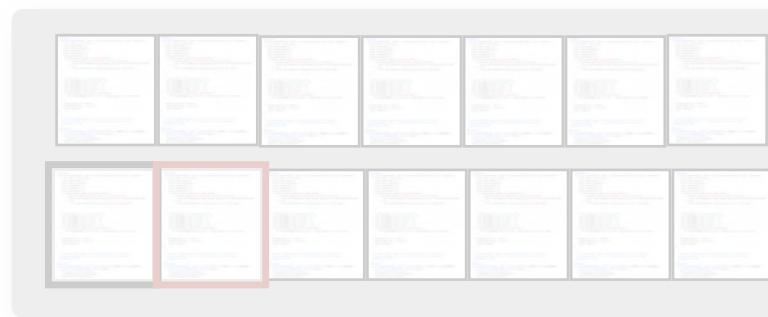
shared  
library



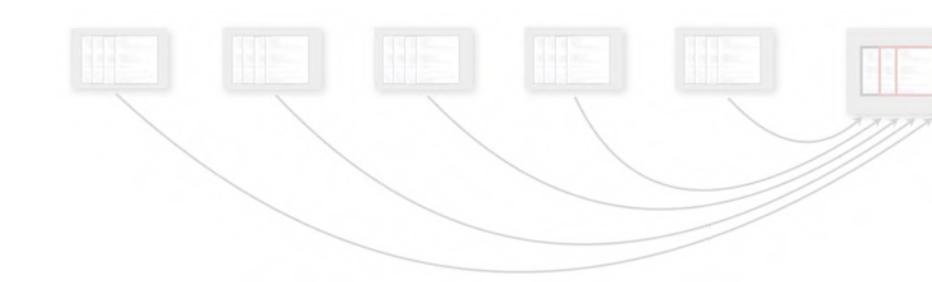
# code sharing techniques



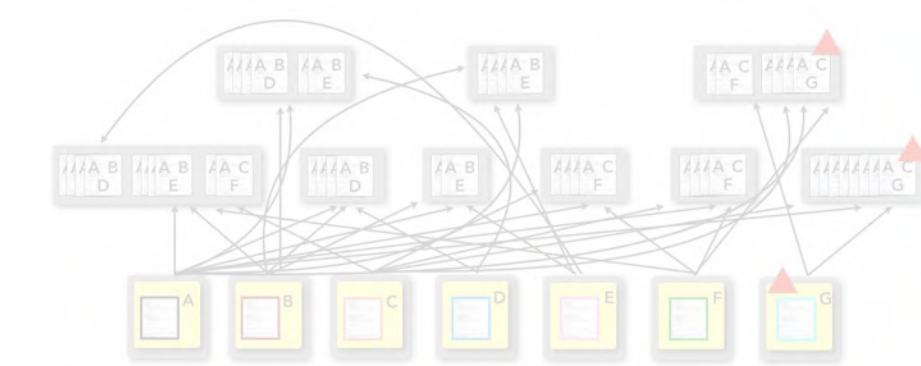
replicated  
code



consolidate  
services



shared  
service

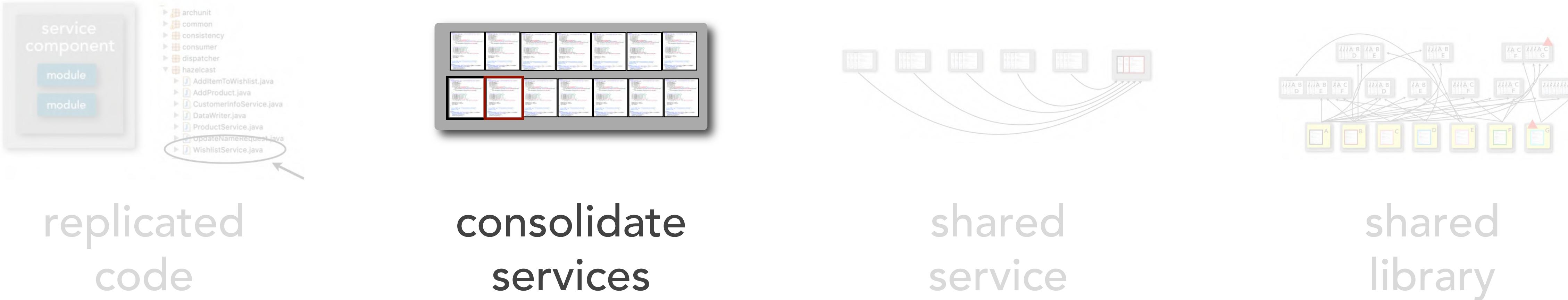


shared  
library



shared code that doesn't change (static code)

# code sharing techniques



minimal services (2-3) dependent on shared code  
that changes frequently

# code sharing techniques

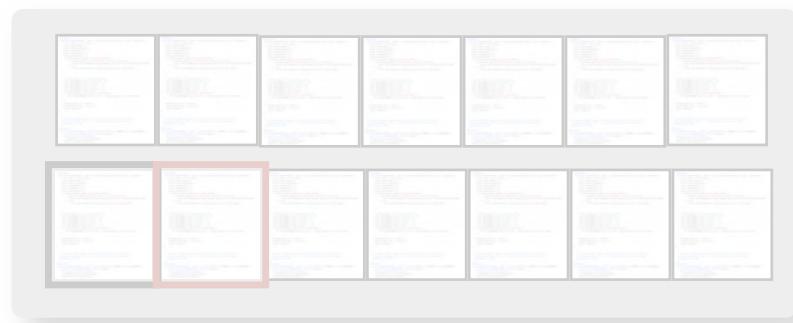


shared code in a heterogeneous environment, or  
shared code in homogeneous environment that  
frequently changes

# code sharing techniques



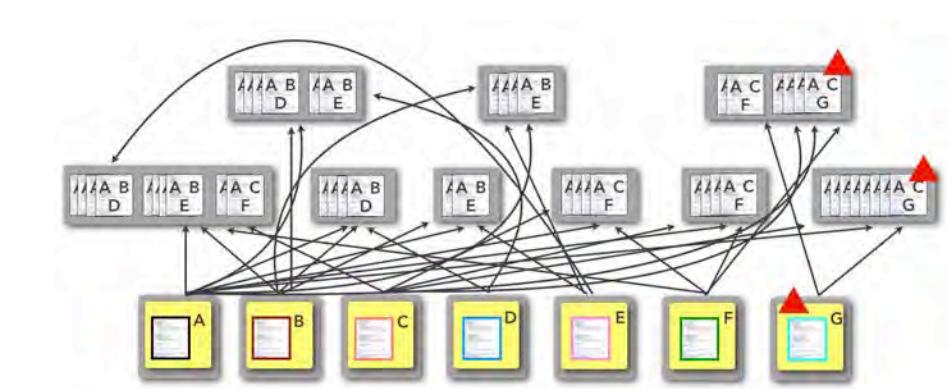
replicated  
code



consolidate  
services



shared  
service



shared  
library

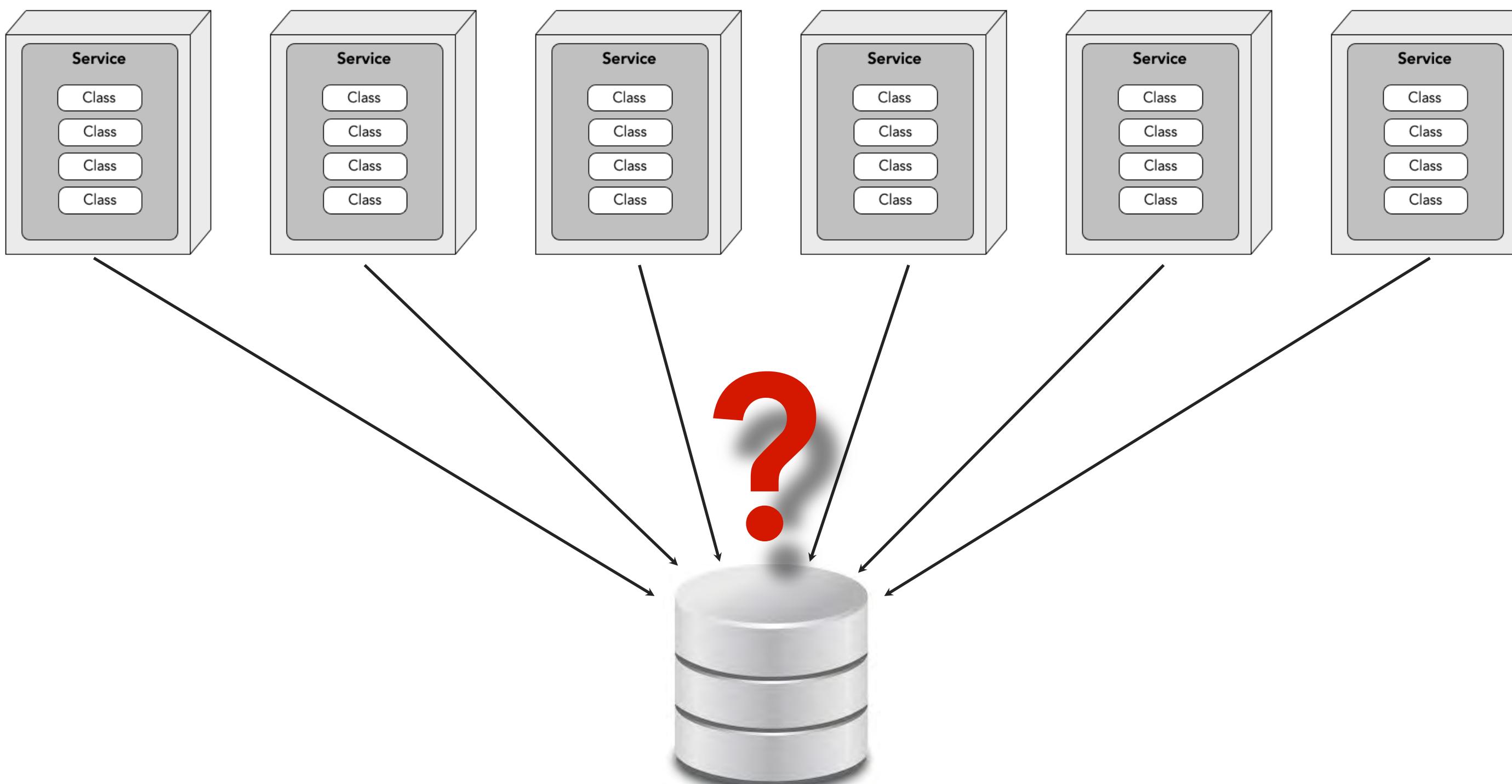


shared code in a homogeneous environment that sometimes changes

# Decomposing Data

# breaking apart data

*“when should I consider breaking apart my data?”*



# breaking apart data

*“when should I consider breaking apart my data?”*

**database granularity drivers**

# breaking apart data

*“when should I consider breaking apart my data?”*

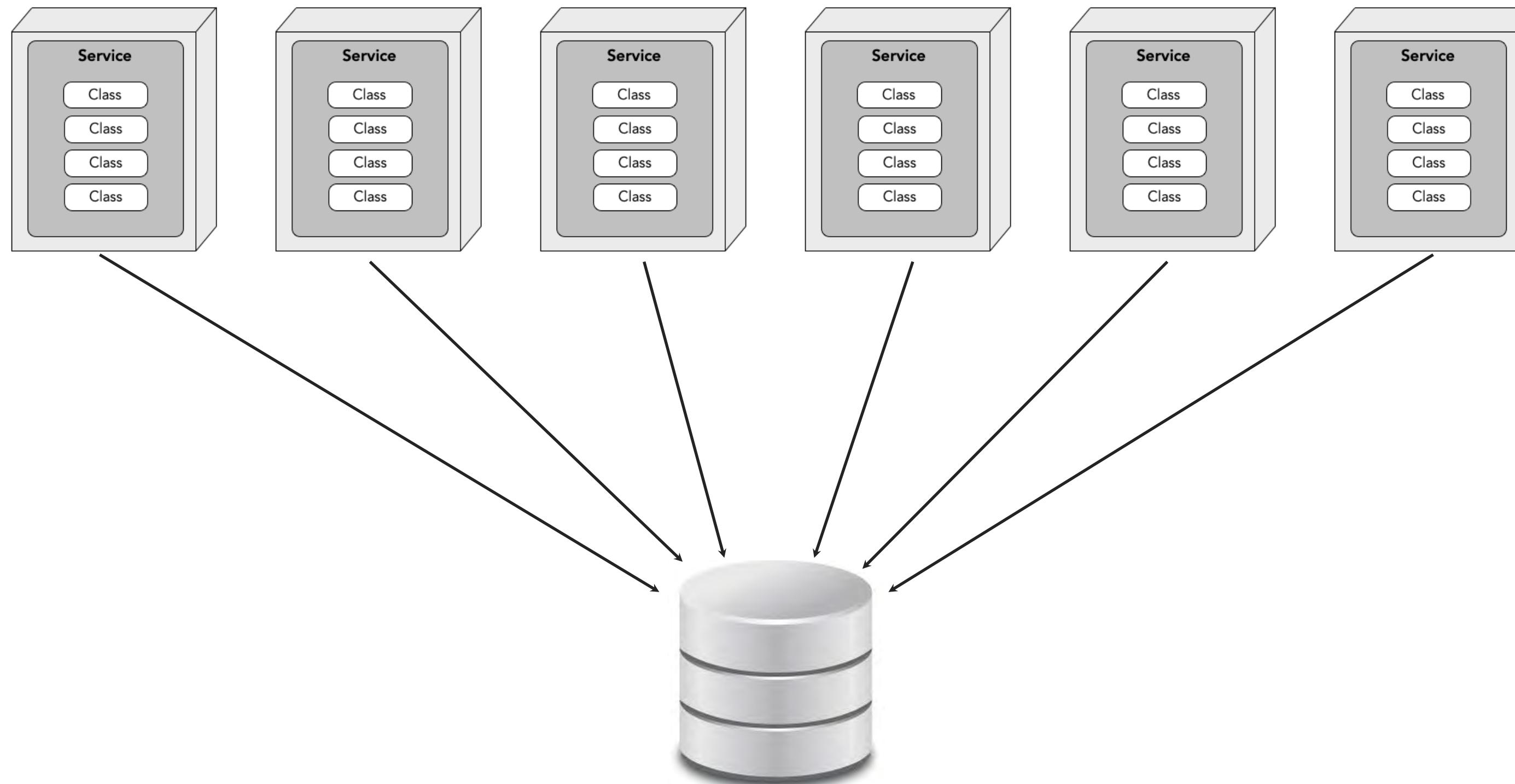
## database granularity drivers



change  
control

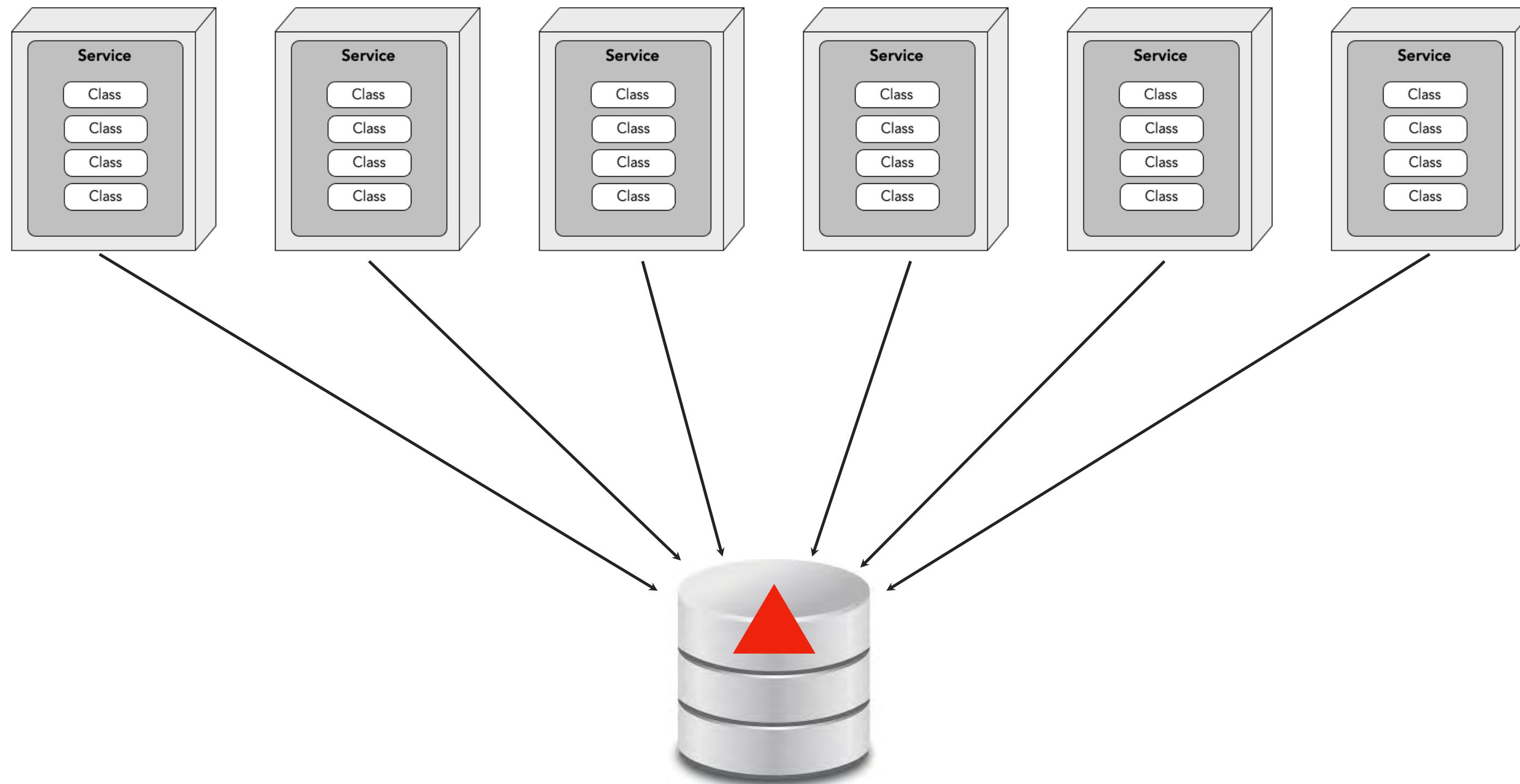
# breaking apart data

## change control



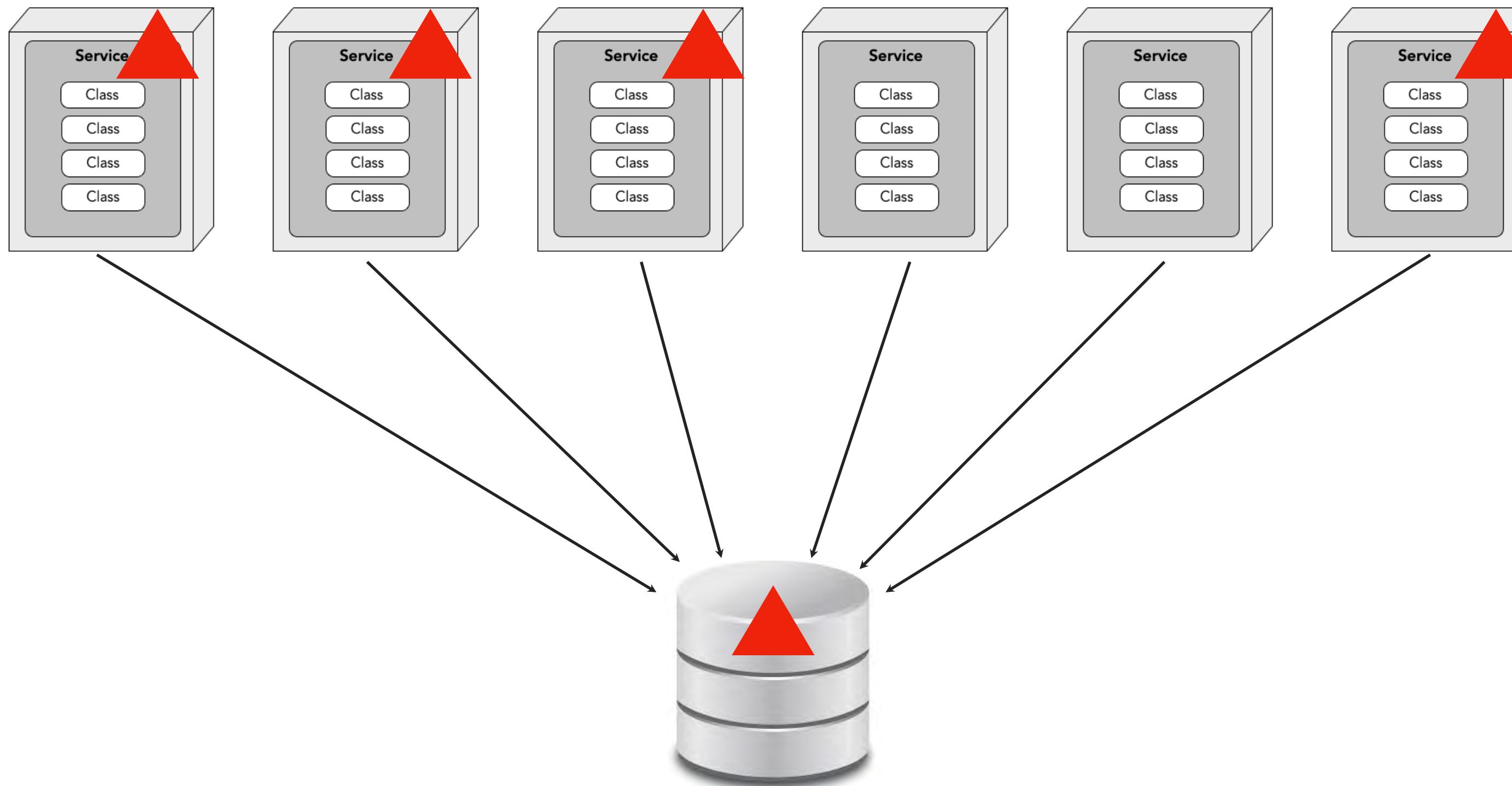
# breaking apart data

## change control



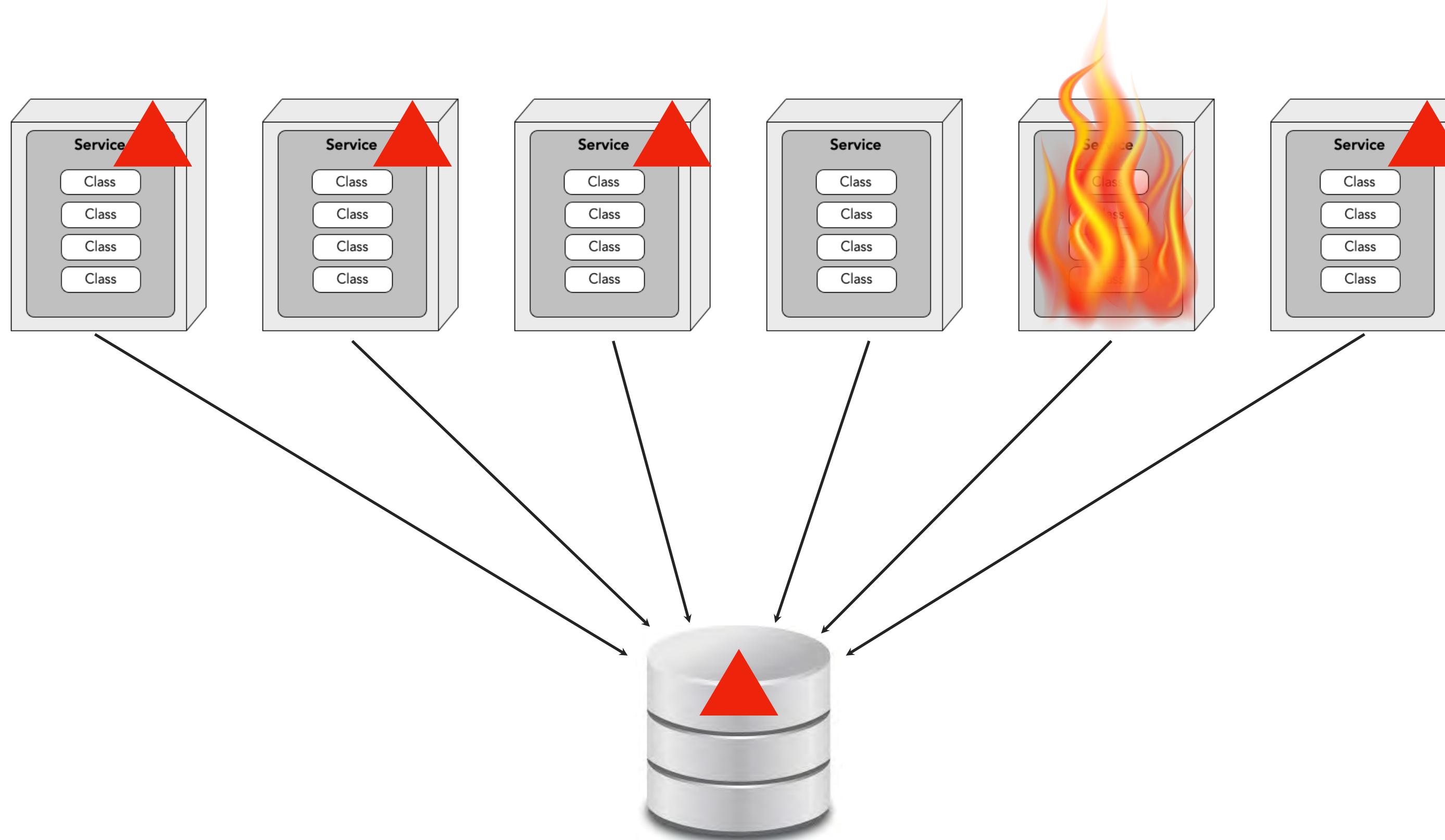
# breaking apart data

## change control



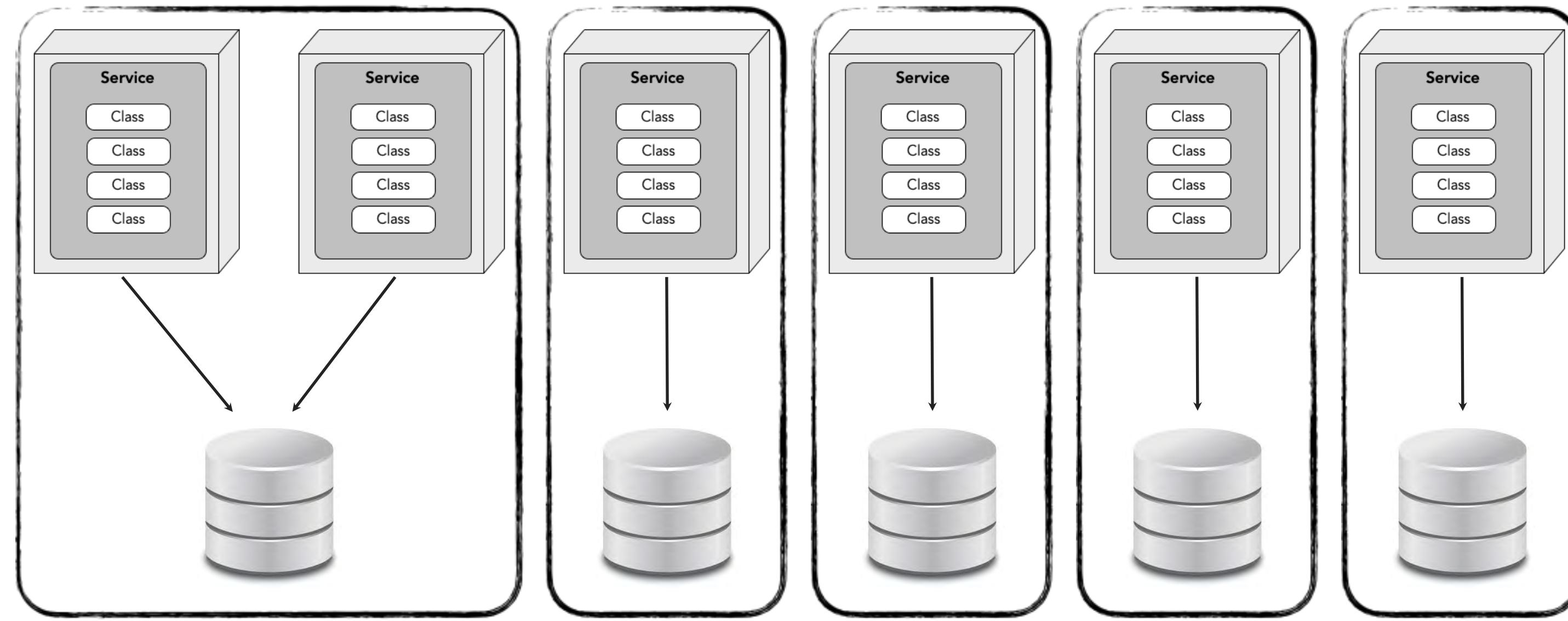
# breaking apart data

## change control



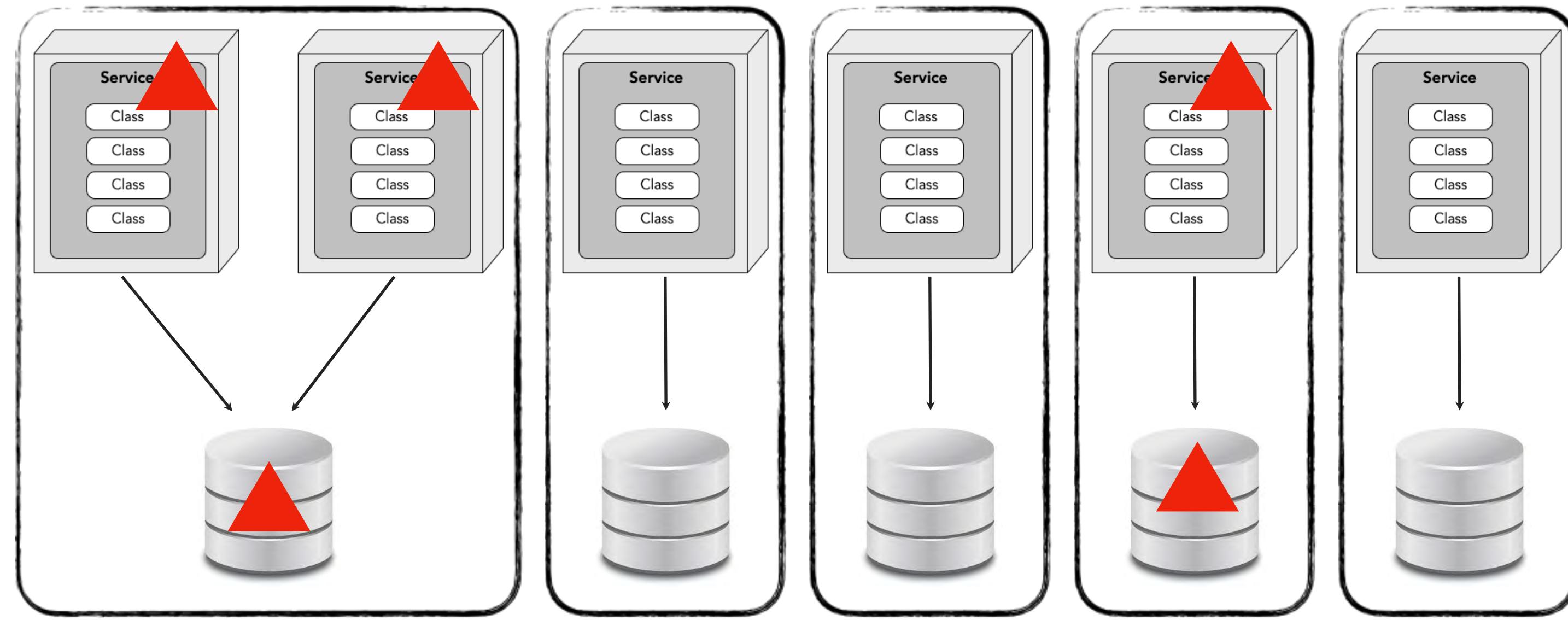
# breaking apart data

## change control



# breaking apart data

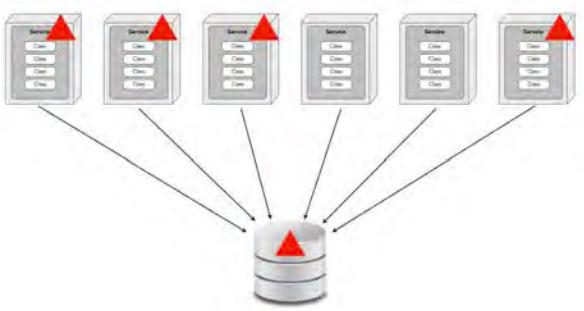
## change control



# breaking apart data

*“when should I consider breaking apart my data?”*

## database granularity drivers

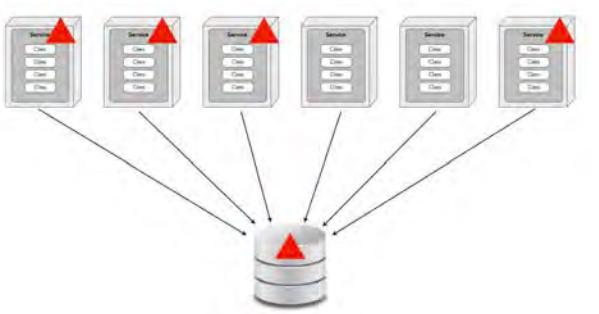


change  
control

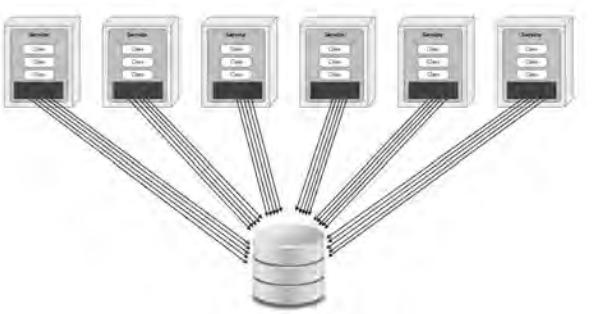
# breaking apart data

*“when should I consider breaking apart my data?”*

## database granularity drivers



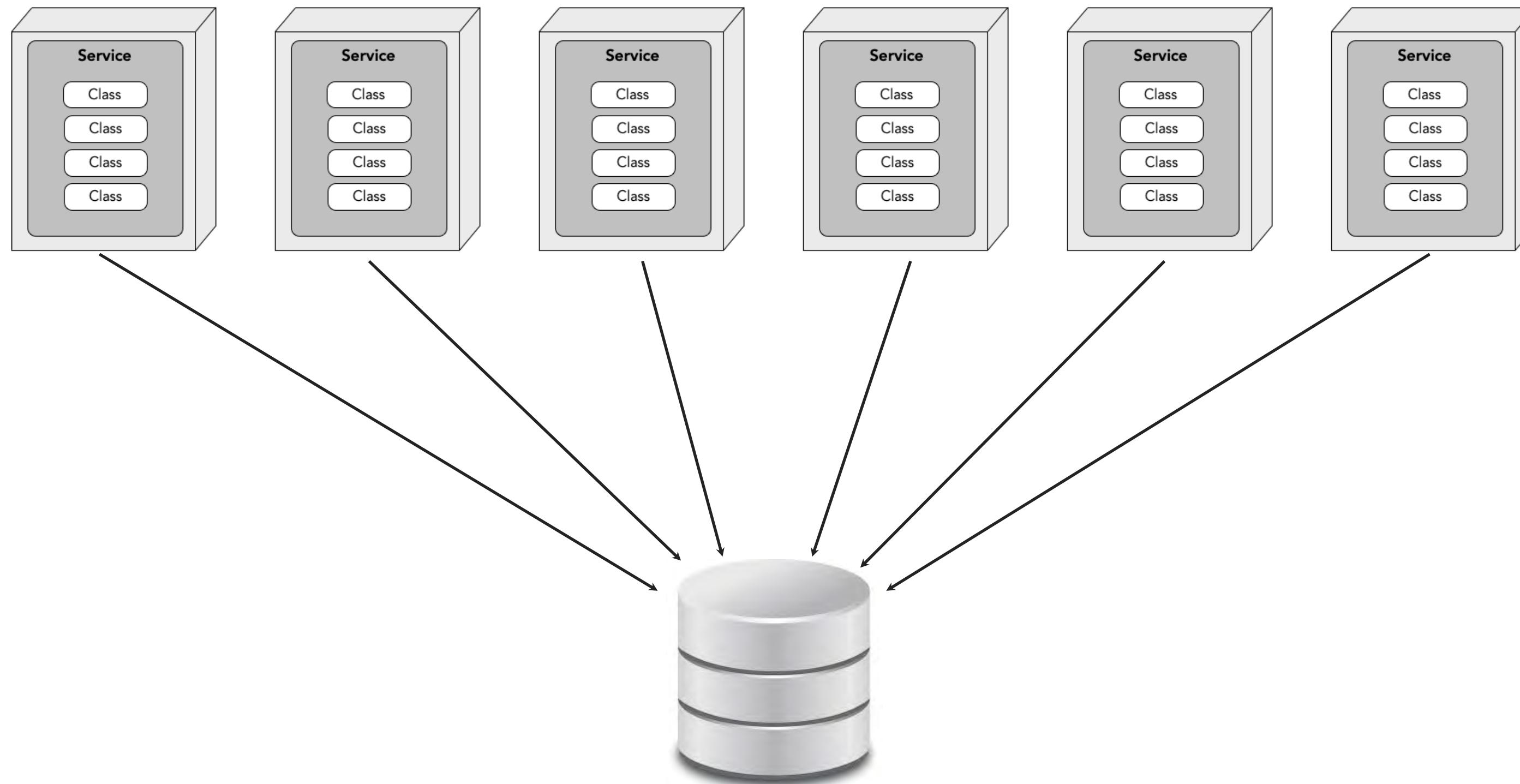
change  
control



database  
connections

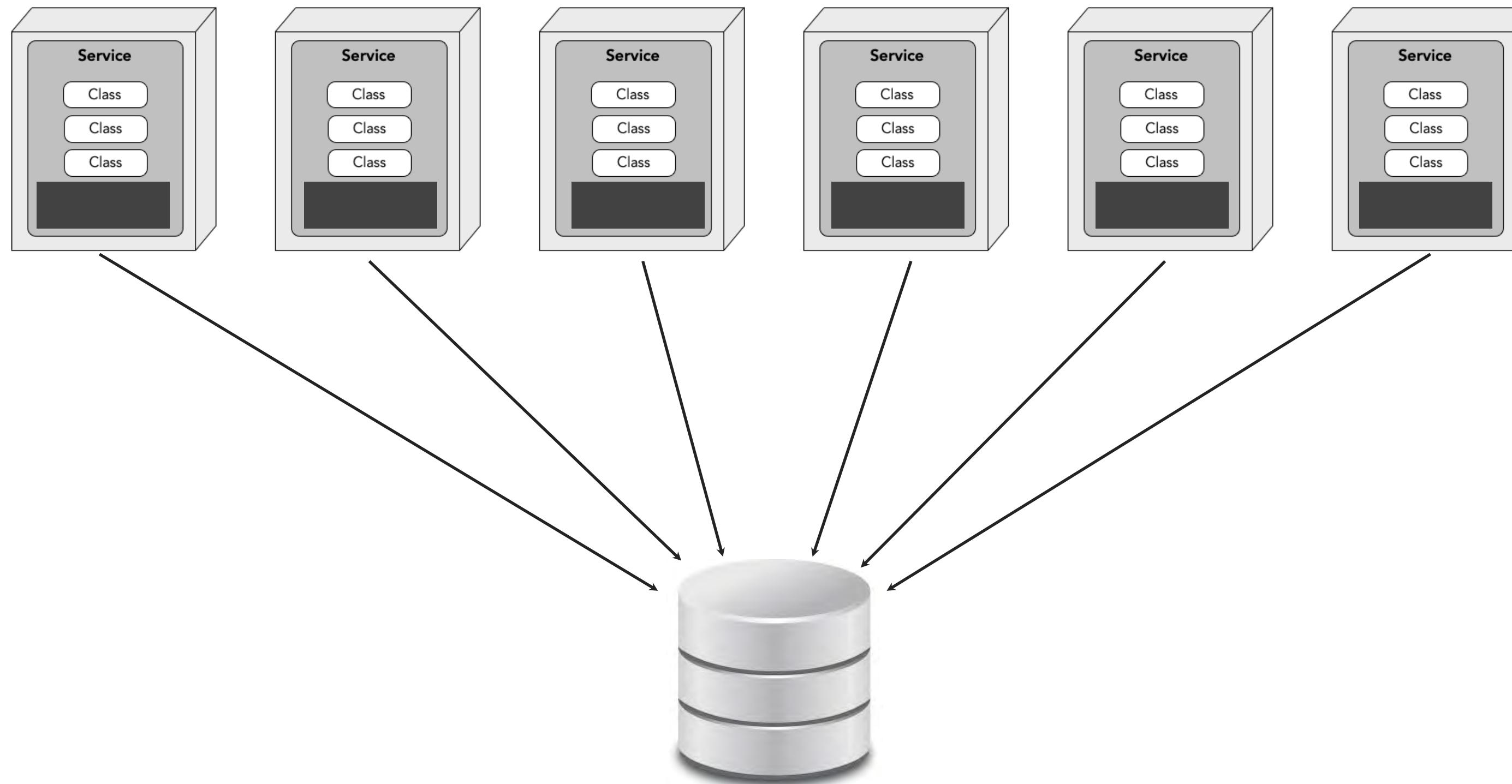
# breaking apart data

## database connections



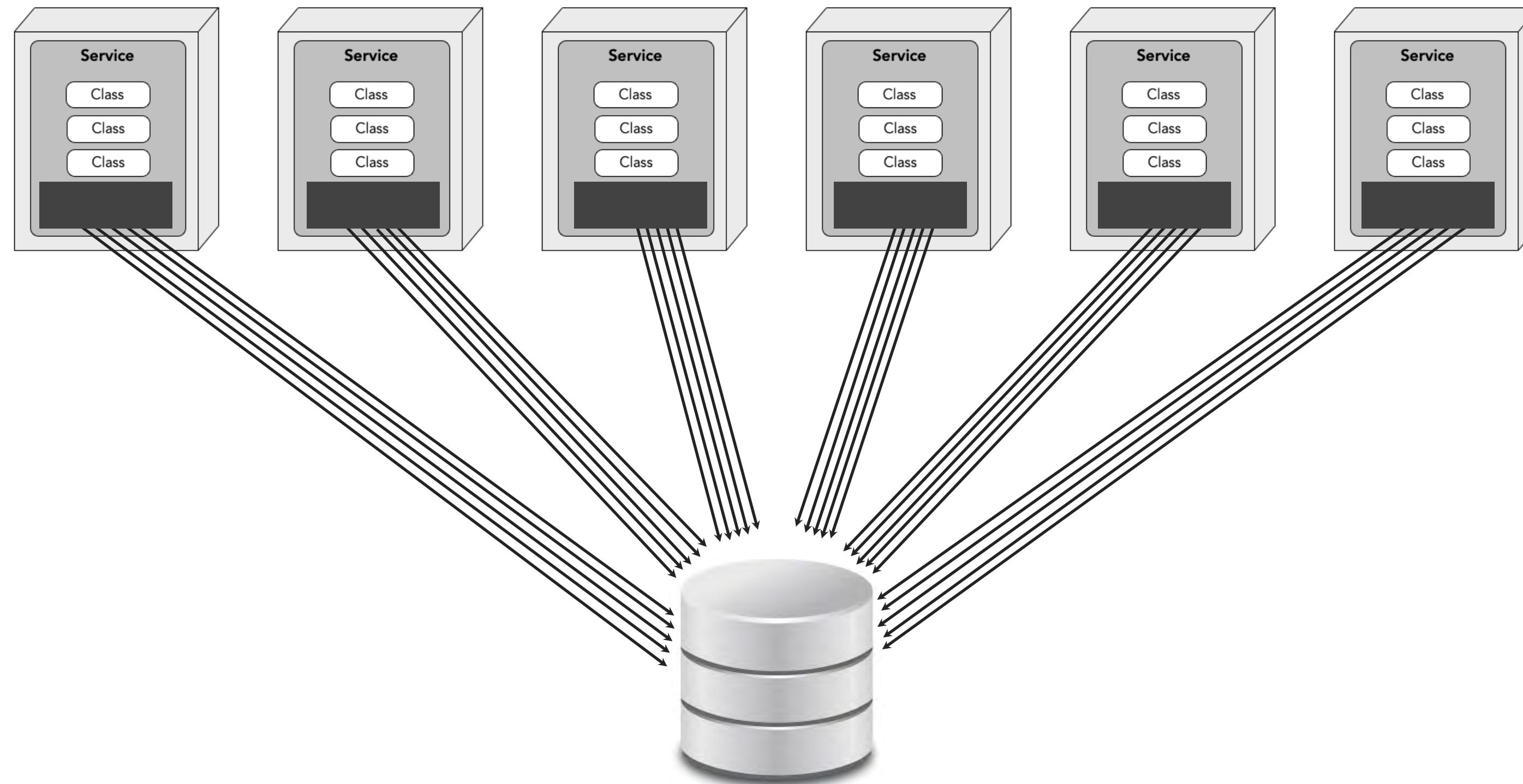
# breaking apart data

## database connections

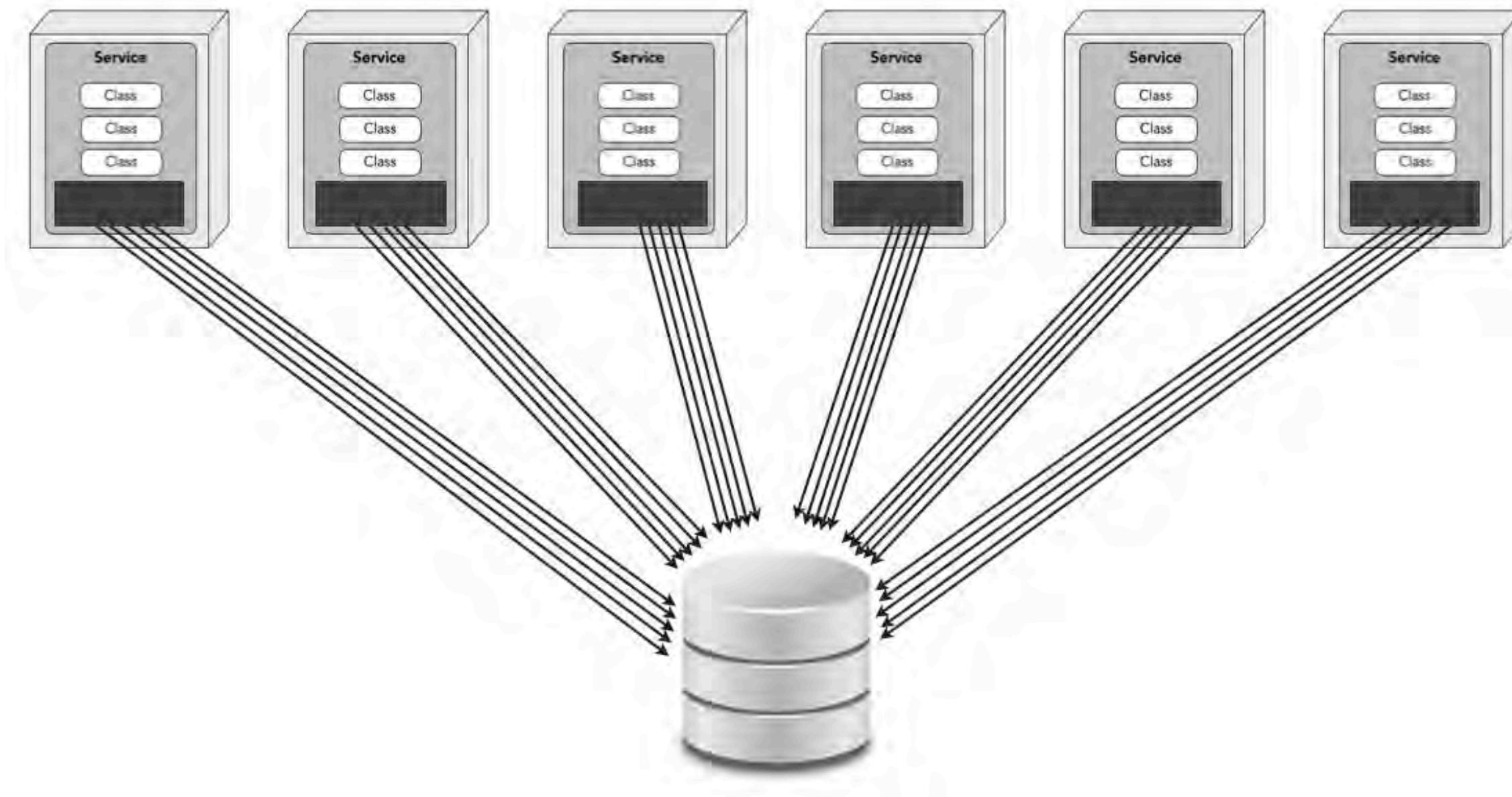


# breaking apart data

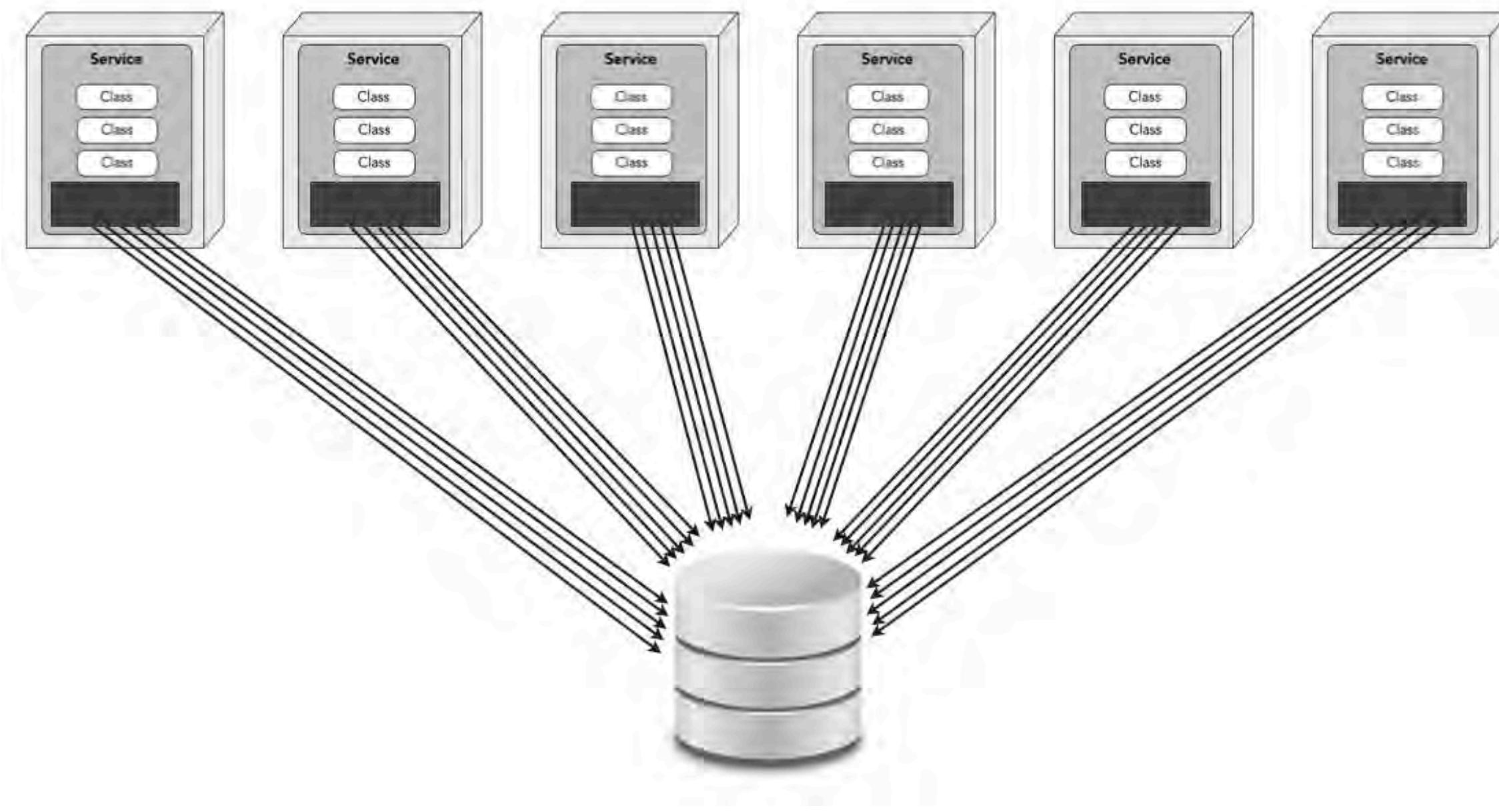
## database connections



# breaking apart data database connections



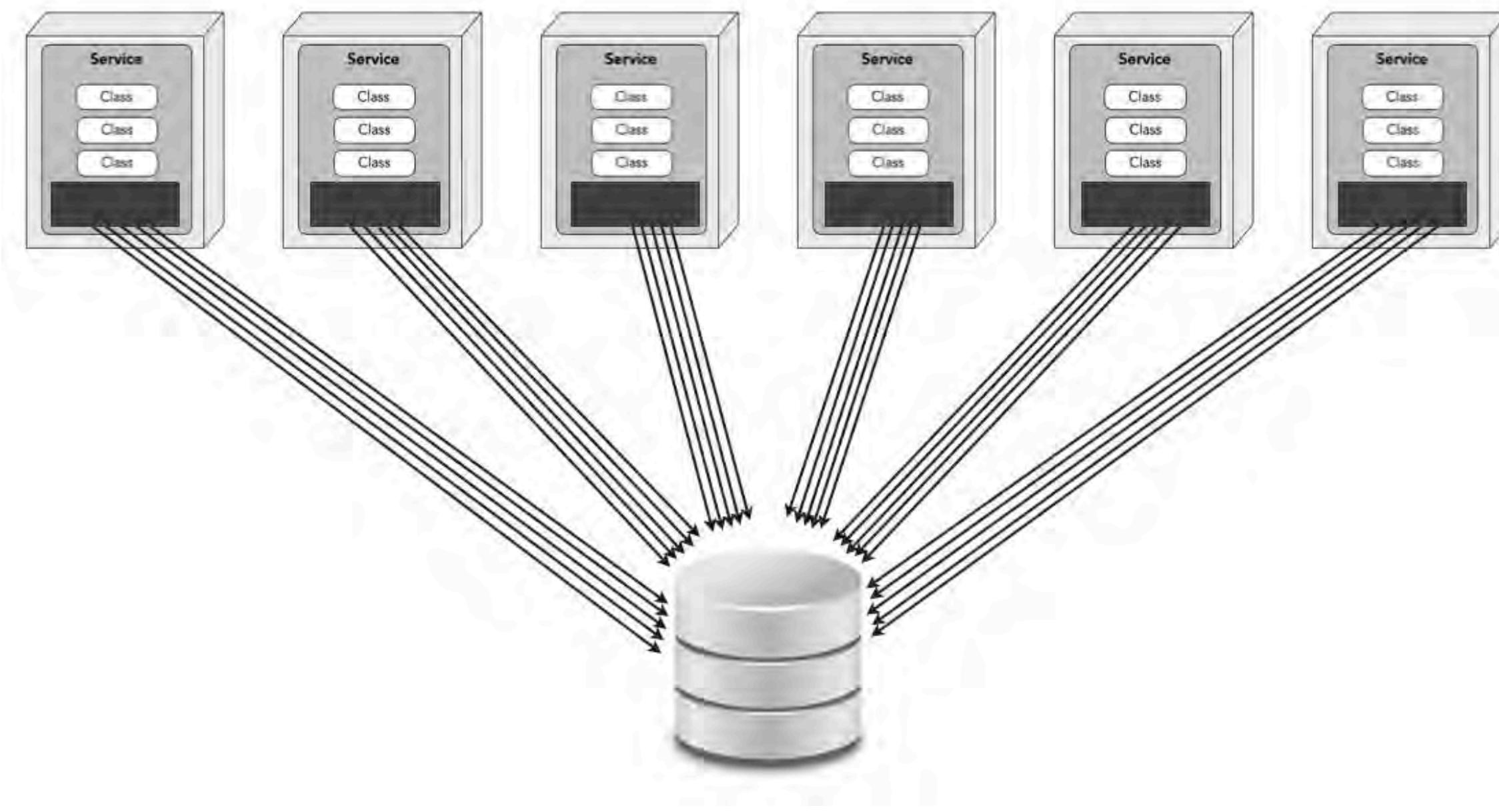
# breaking apart data database connections



monolithic application: 200 connections

# breaking apart data

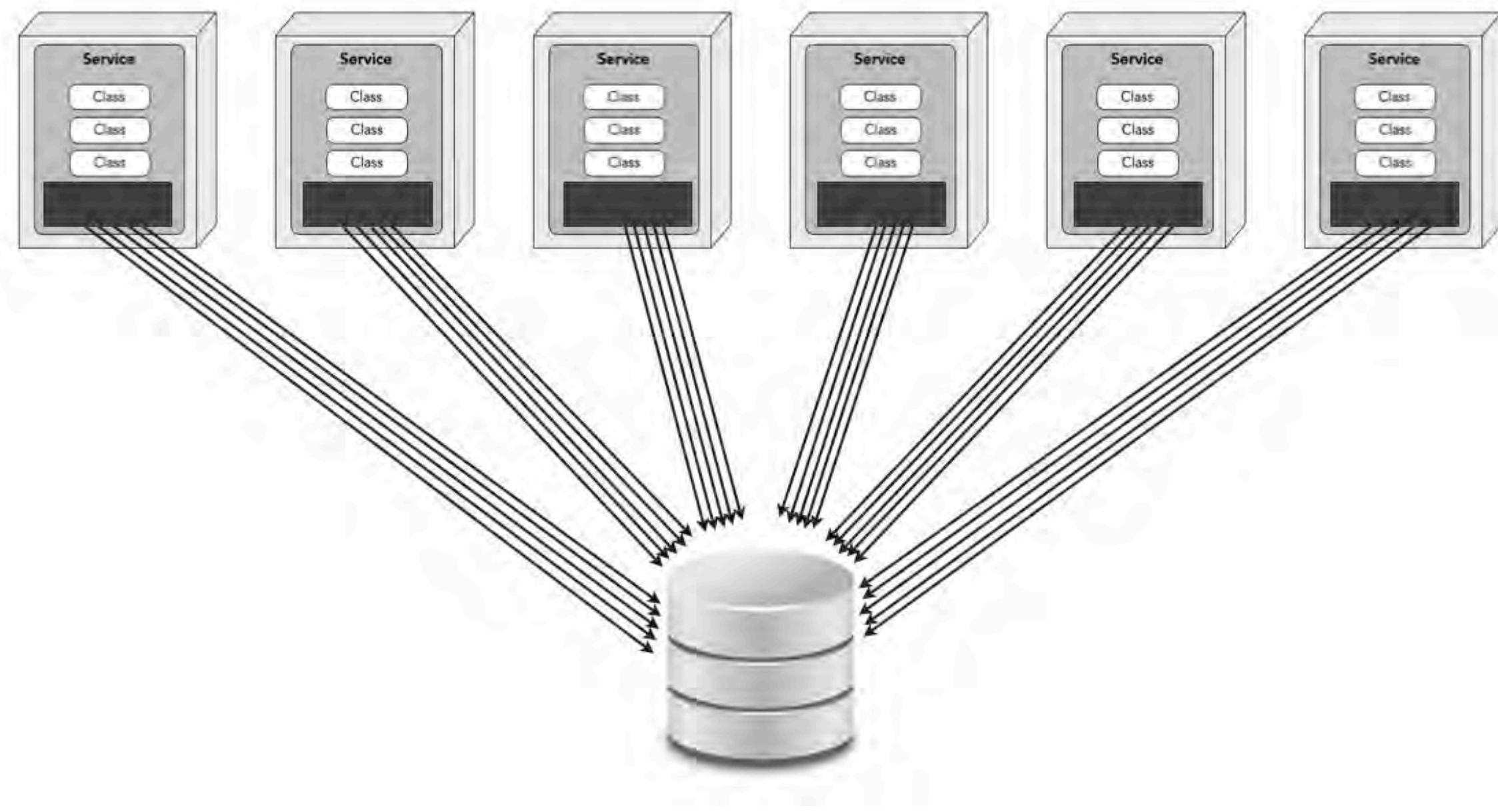
## database connections



monolithic application: 200 connections  
distributed services: 50

# breaking apart data

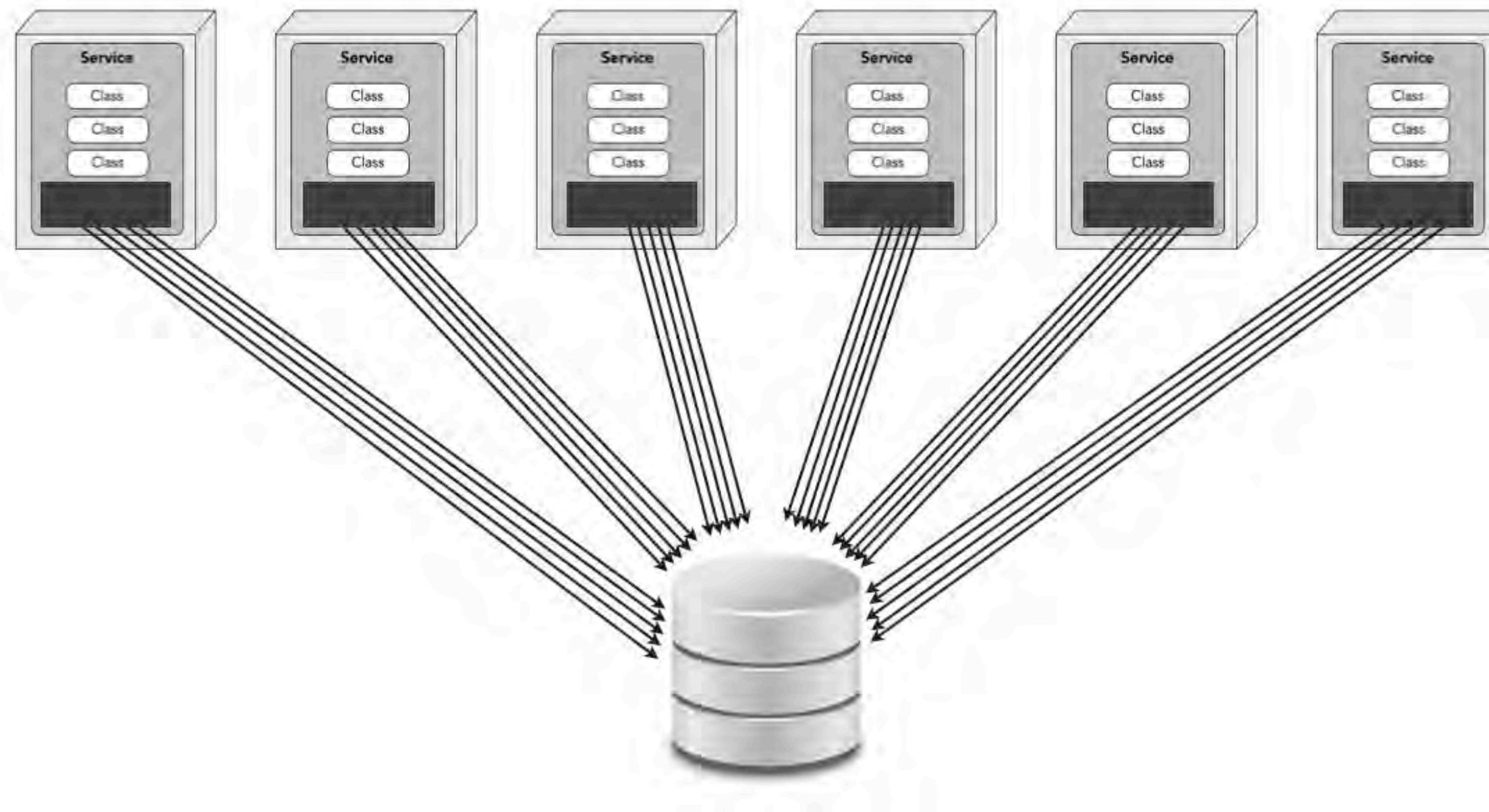
## database connections



monolithic application: 200 connections  
distributed services: 50  
connections per service: 10

# breaking apart data

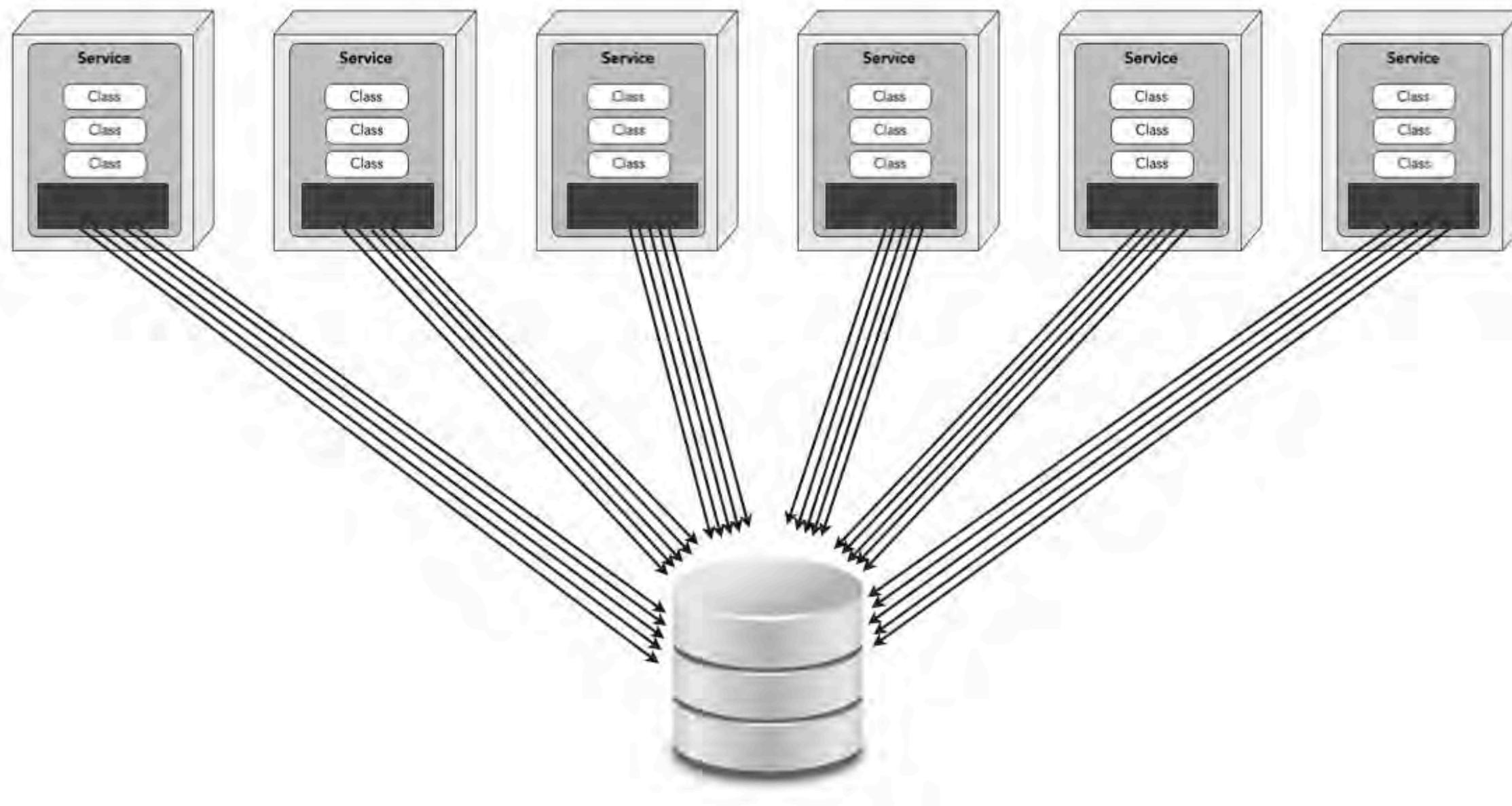
## database connections



monolithic application: 200 connections  
distributed services: 50  
connections per service: 10  
service instances (min): 2

# breaking apart data

## database connections



monolithic application: 200 connections

distributed services: 50

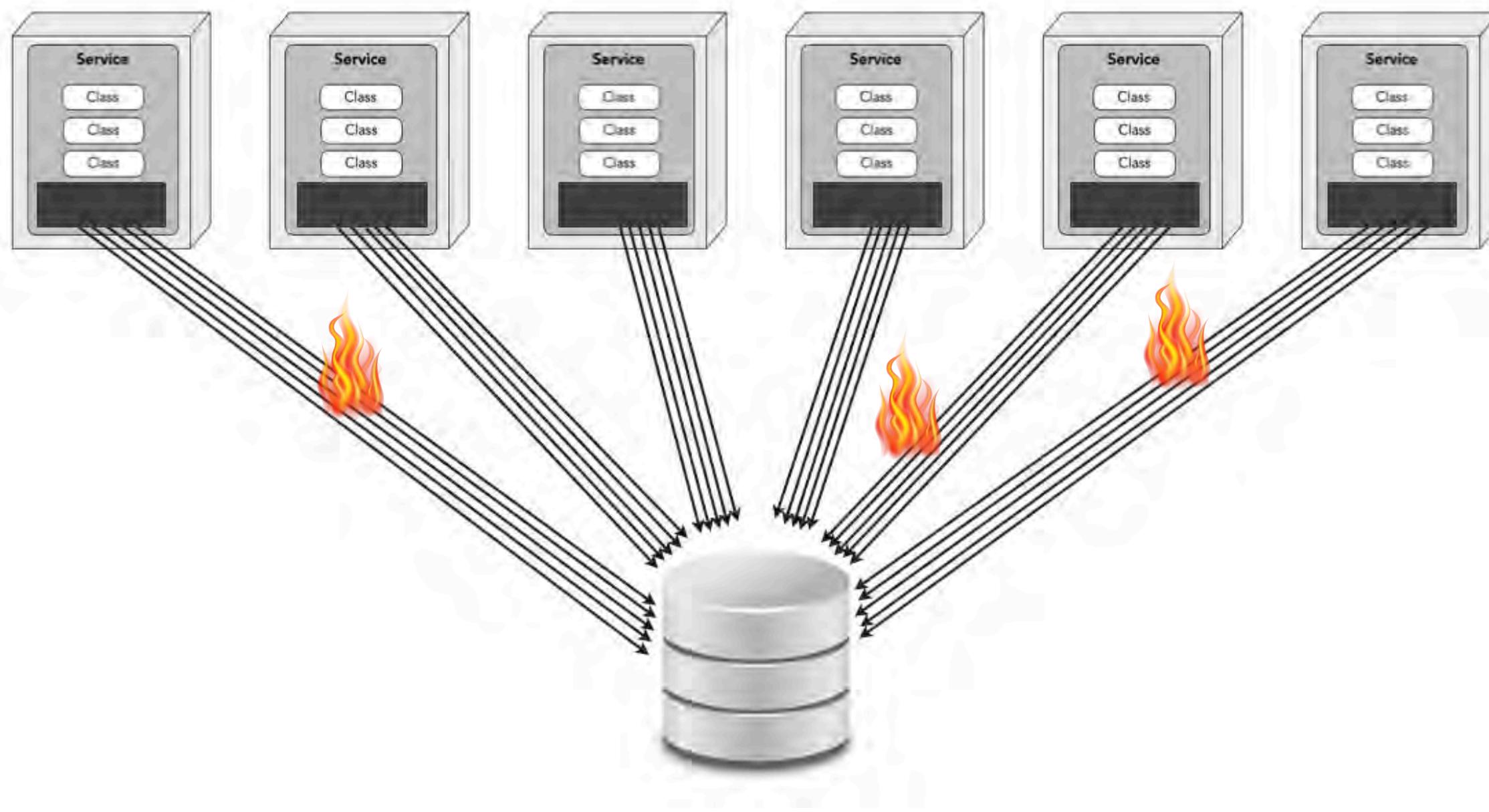
connections per service: 10

service instances (min): 2

**distributed connections: 1000 connections**

# breaking apart data

## database connections



monolithic application: 200 connections

distributed services: 50

connections per service: 10

service instances (min): 2

**distributed connections: 1000 connections**

# breaking apart data

*“when should I consider breaking apart my data?”*

## database granularity drivers



change  
control

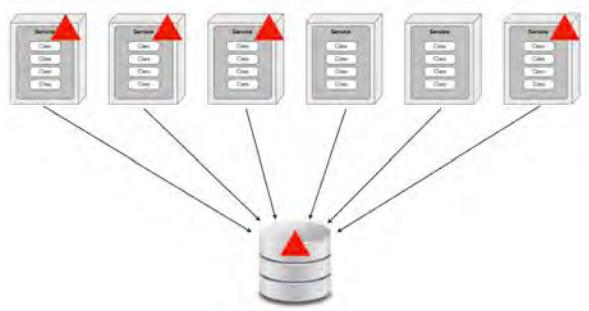


database  
connections

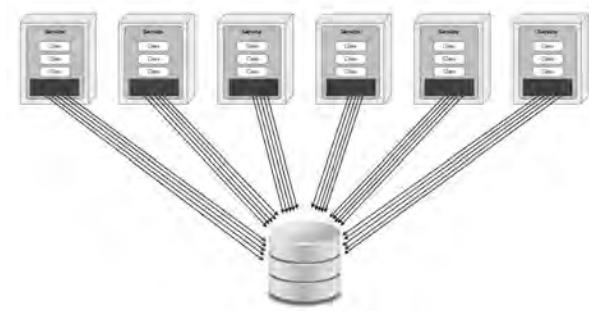
# breaking apart data

*“when should I consider breaking apart my data?”*

## database granularity drivers



change  
control



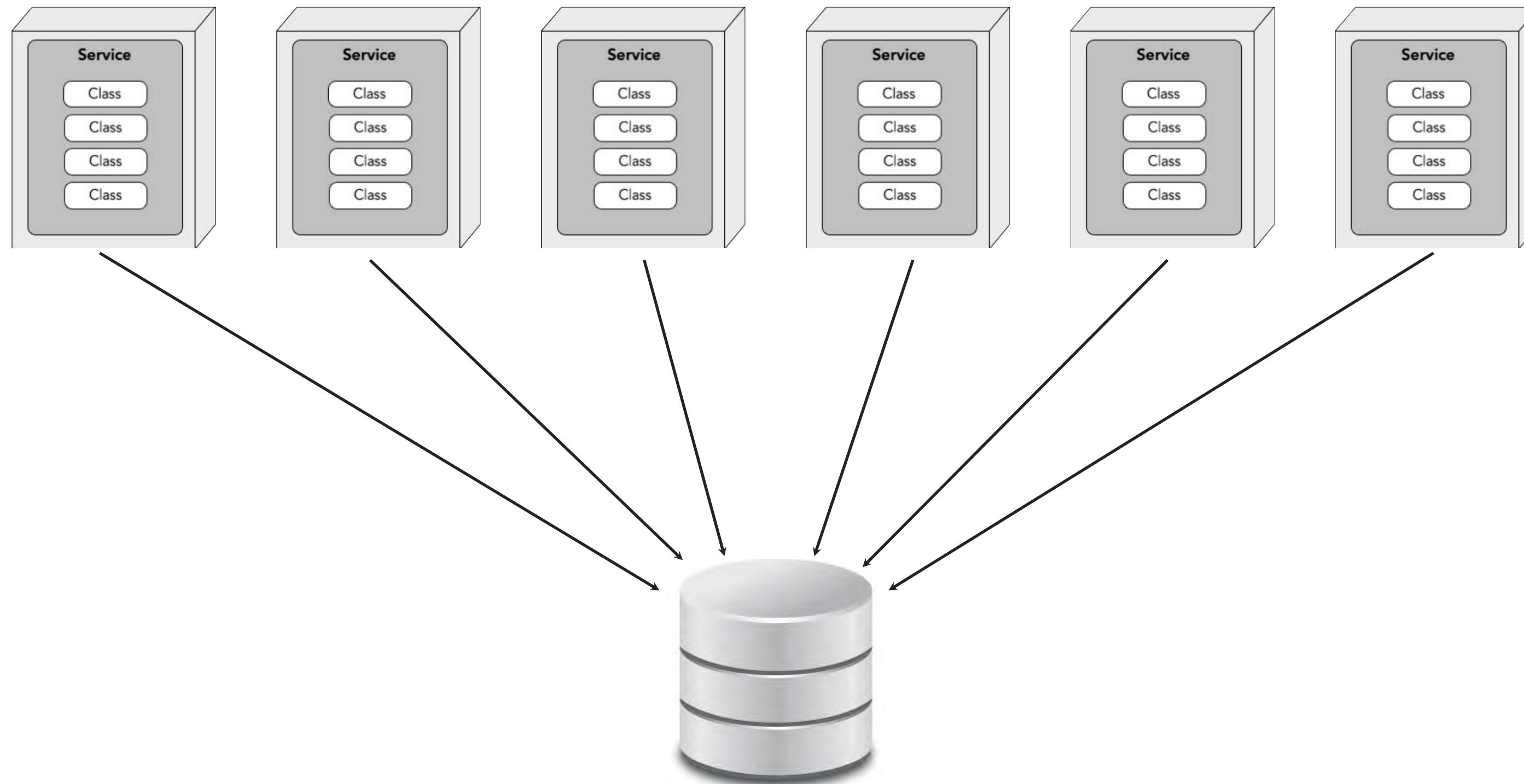
database  
connections



database  
scalability

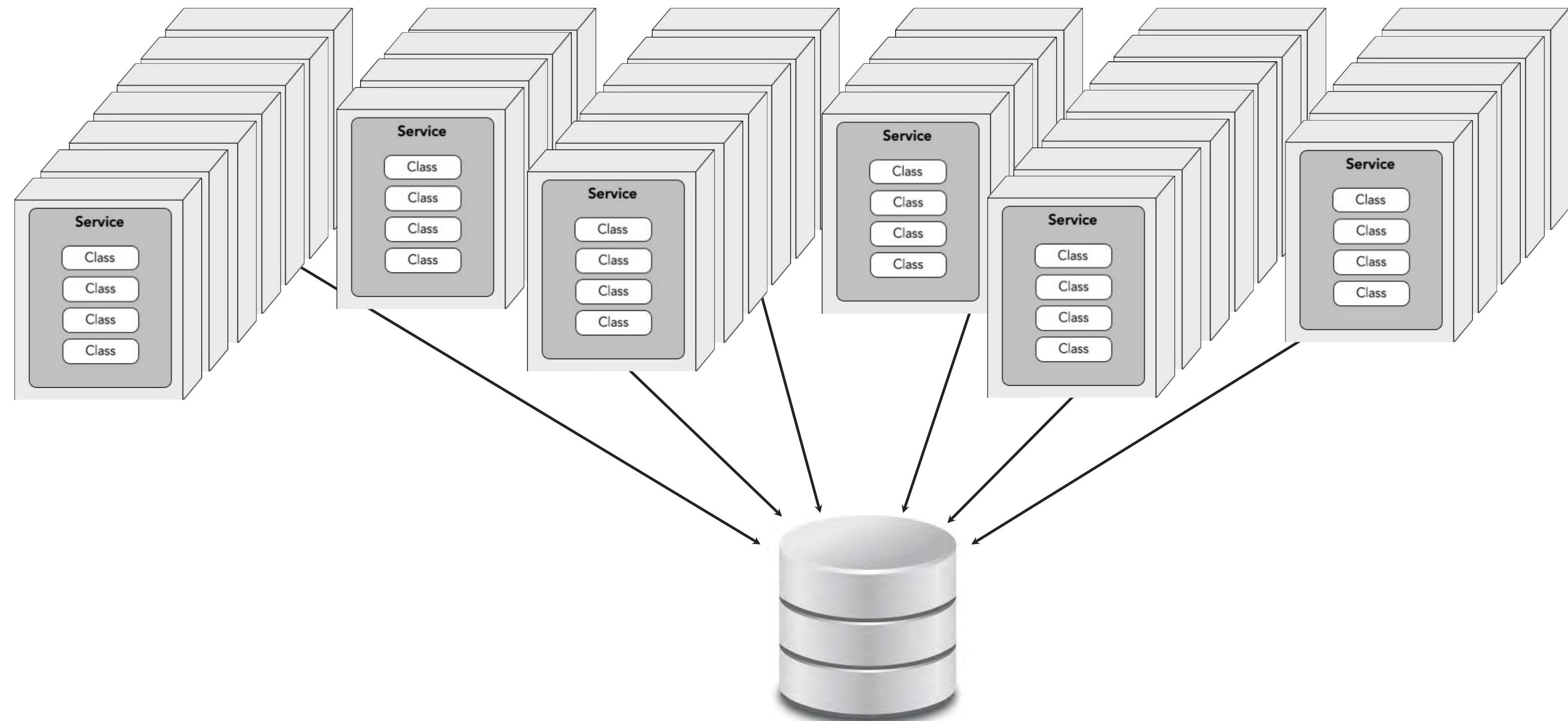
# breaking apart data

## database scalability



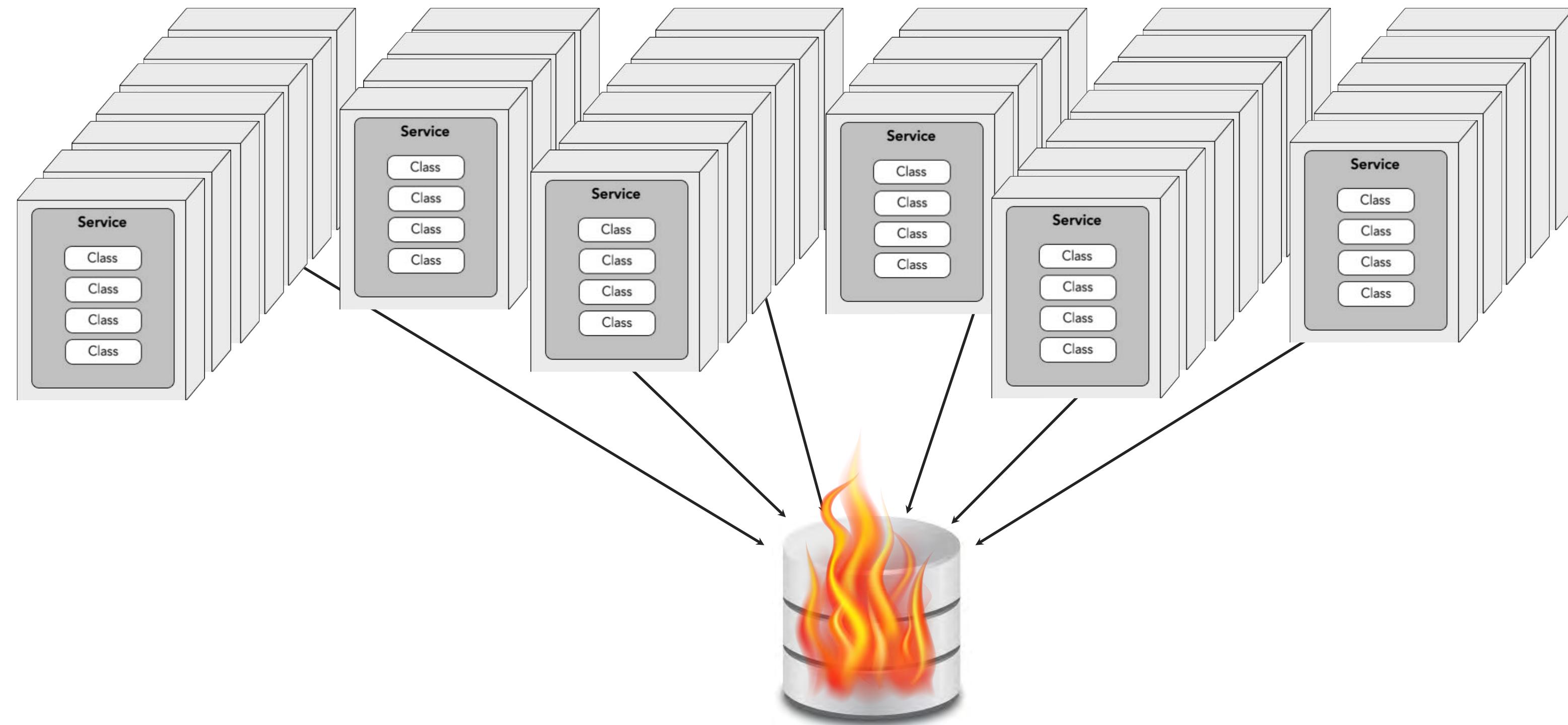
# breaking apart data

## database scalability



# breaking apart data

## database scalability



# breaking apart data

*“when should I consider breaking apart my data?”*

## database granularity drivers



change  
control



database  
connections

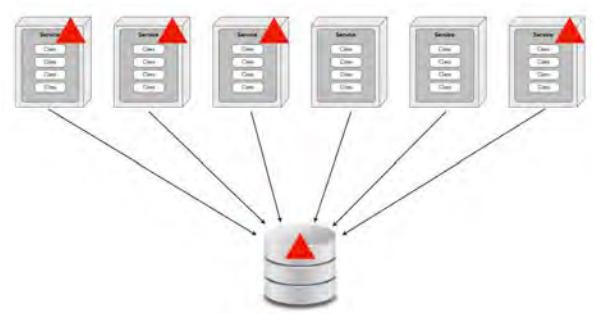


database  
scalability

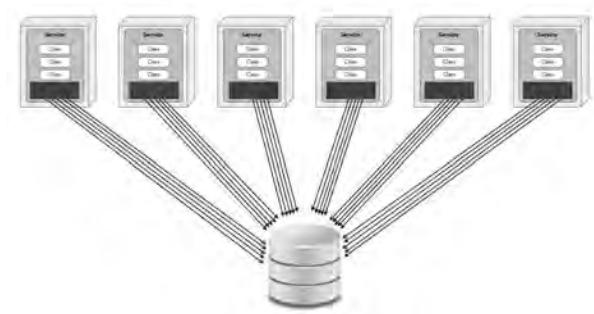
# breaking apart data

*“when should I consider breaking apart my data?”*

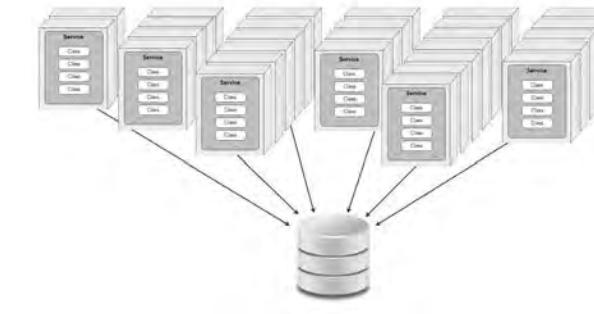
## database granularity drivers



change  
control



database  
connections



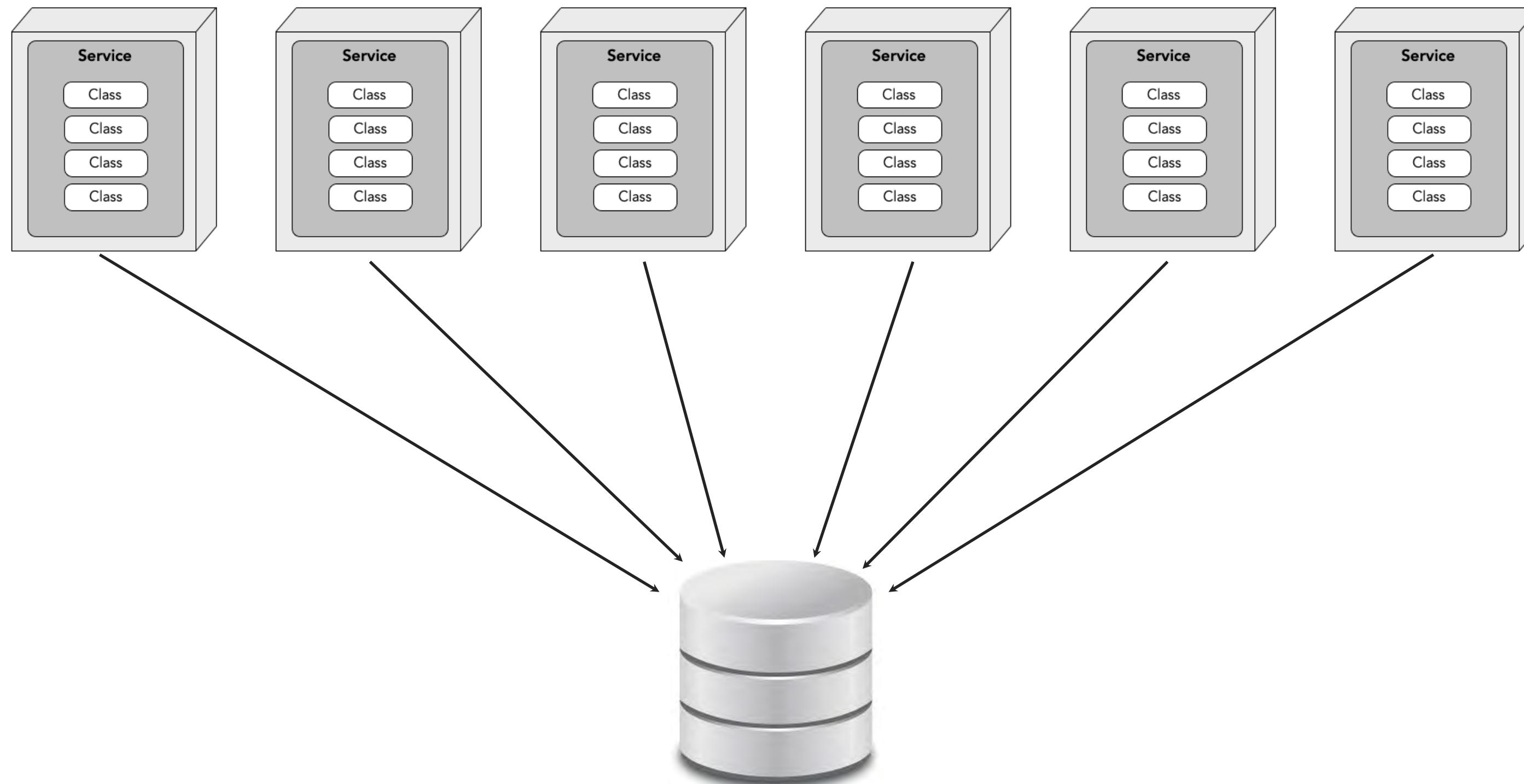
database  
scalability



fault  
tolerance

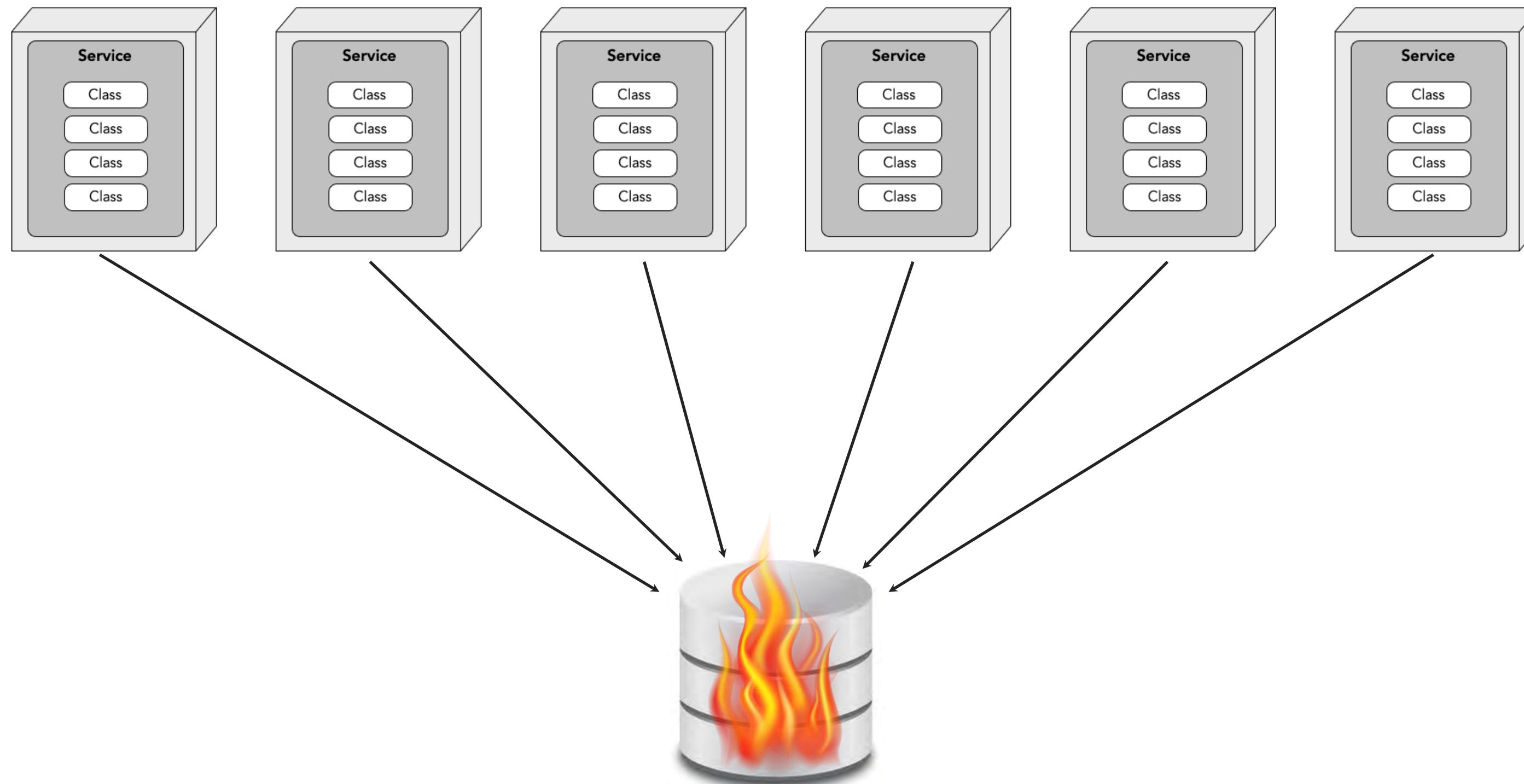
# breaking apart data

## fault tolerance



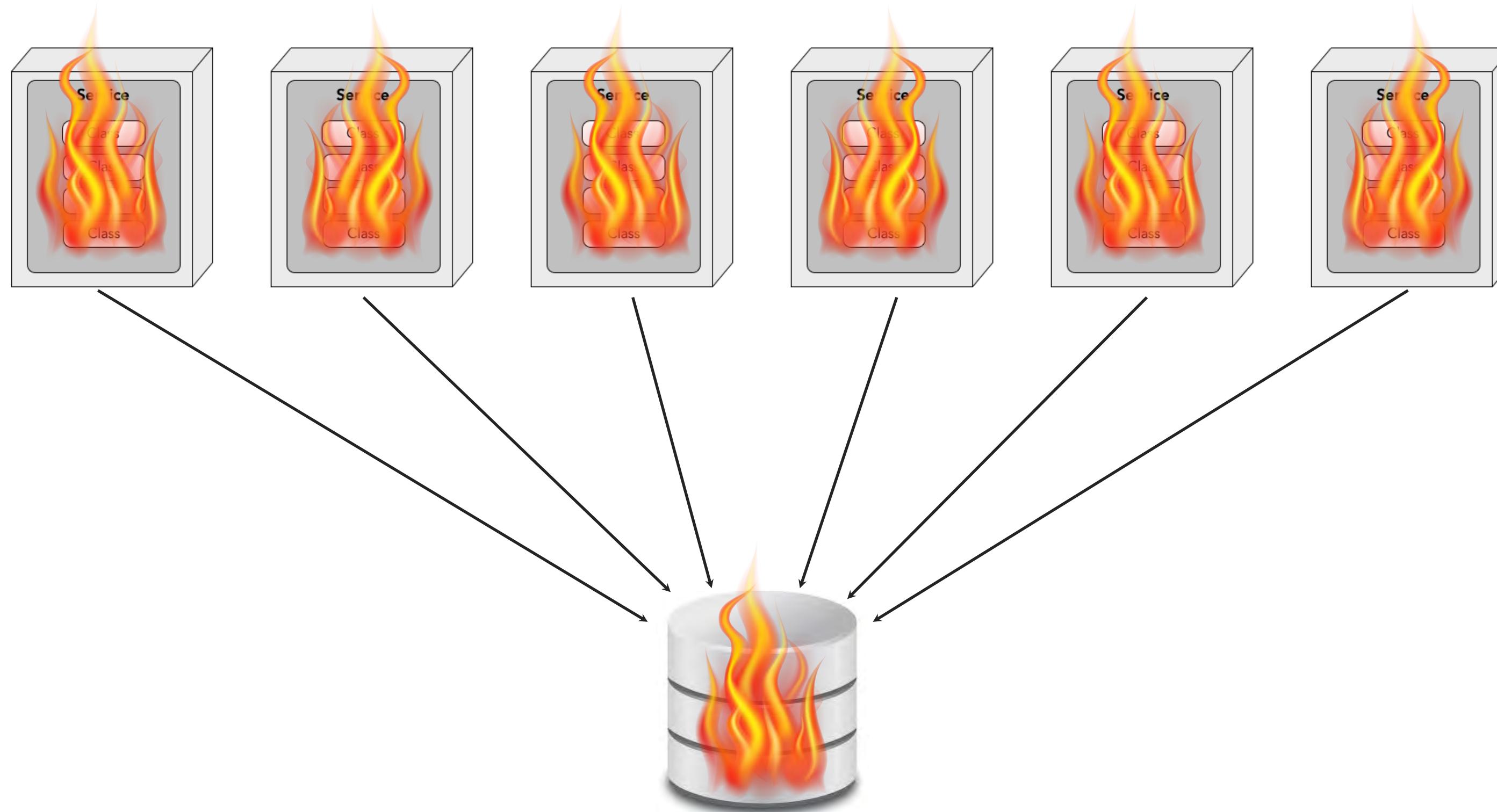
# breaking apart data

## fault tolerance



# breaking apart data

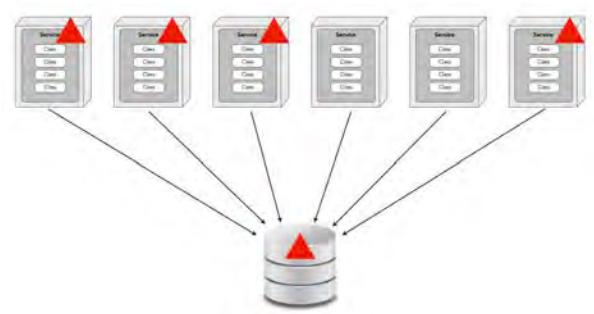
## fault tolerance



# breaking apart data

*“when should I consider breaking apart my data?”*

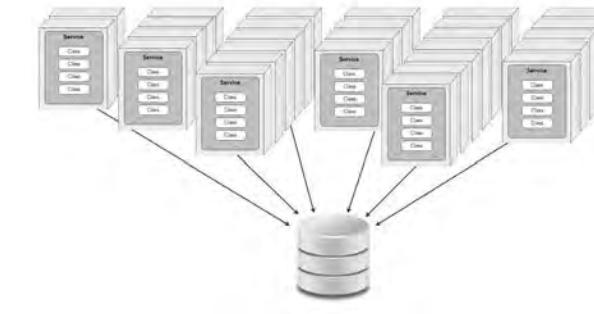
## database granularity drivers



change  
control



database  
connections

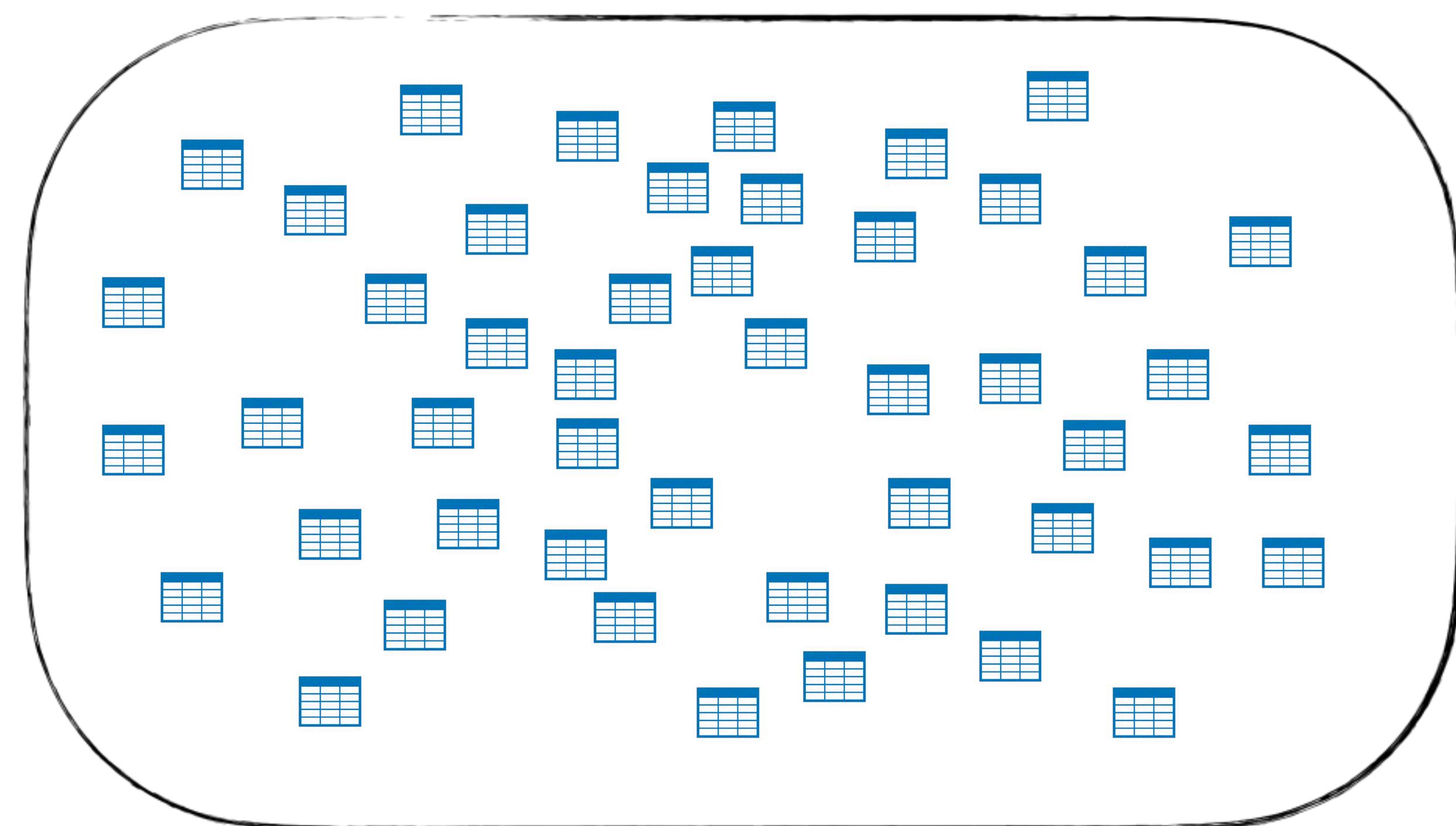


database  
scalability

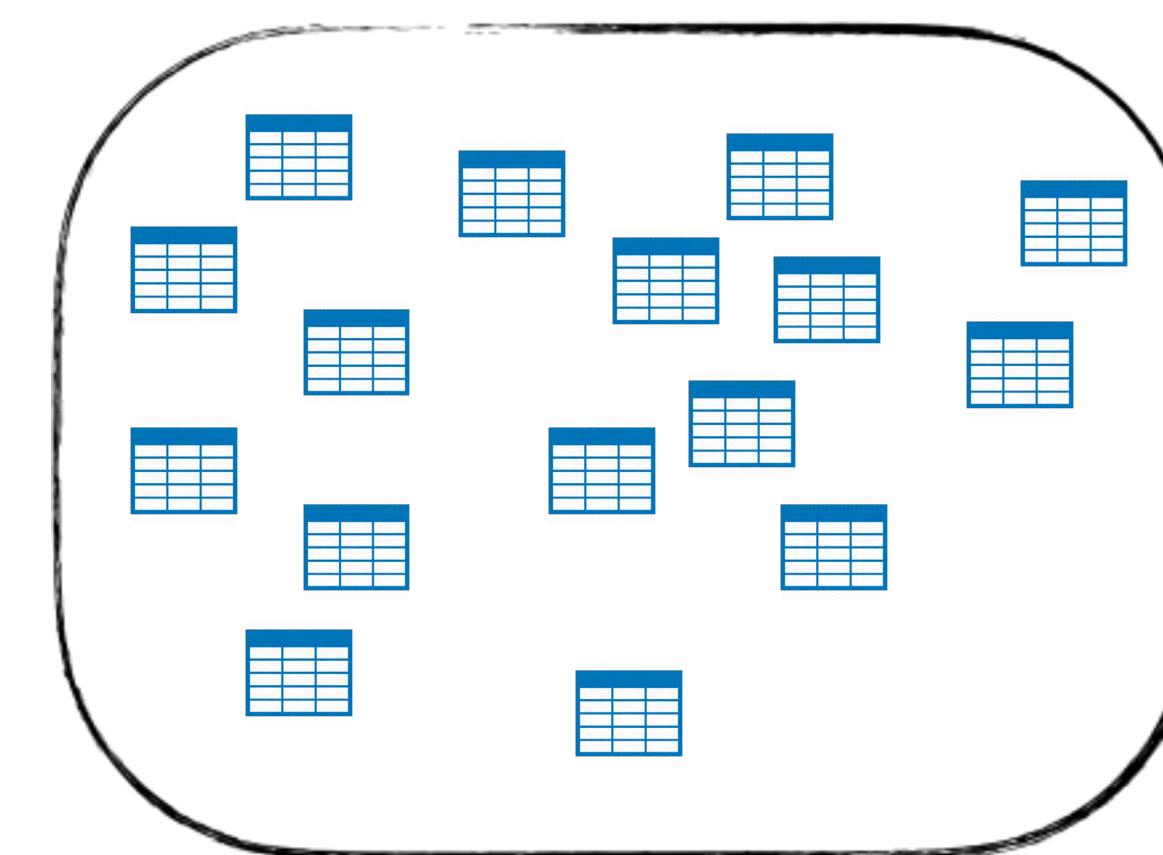
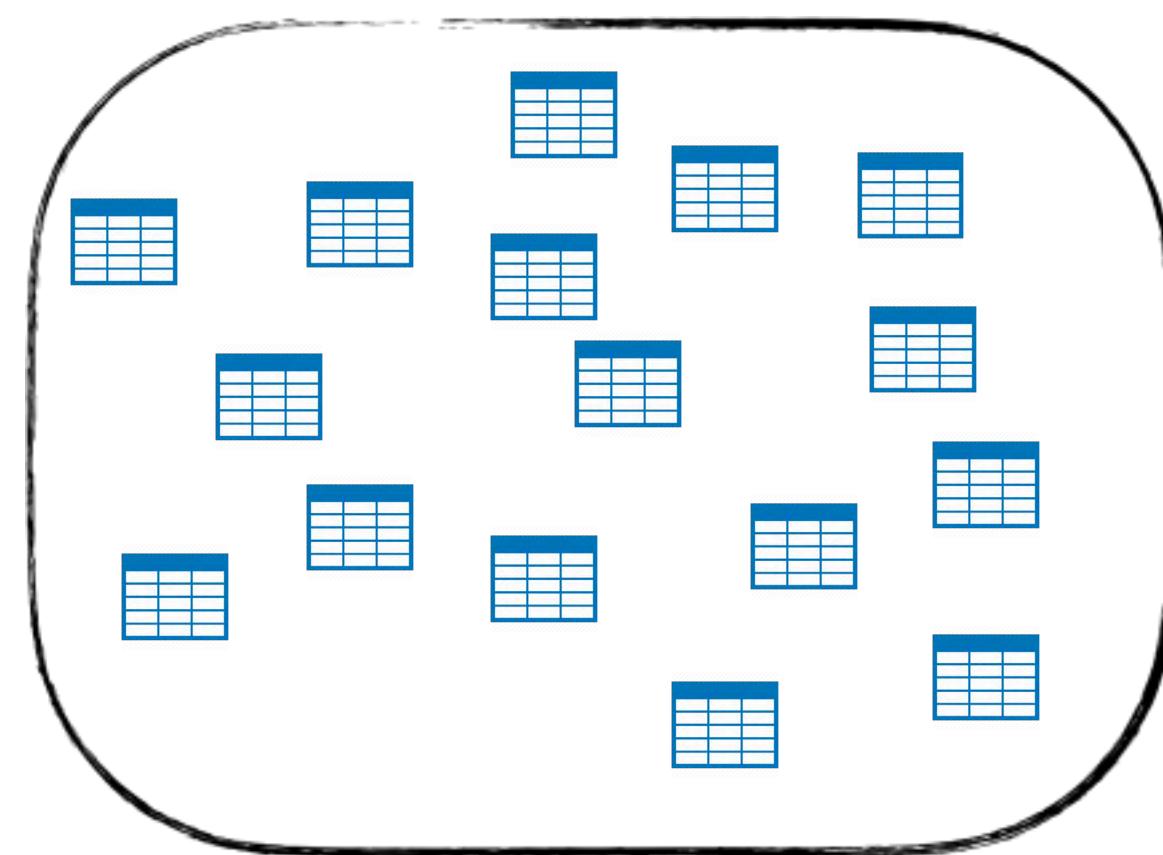
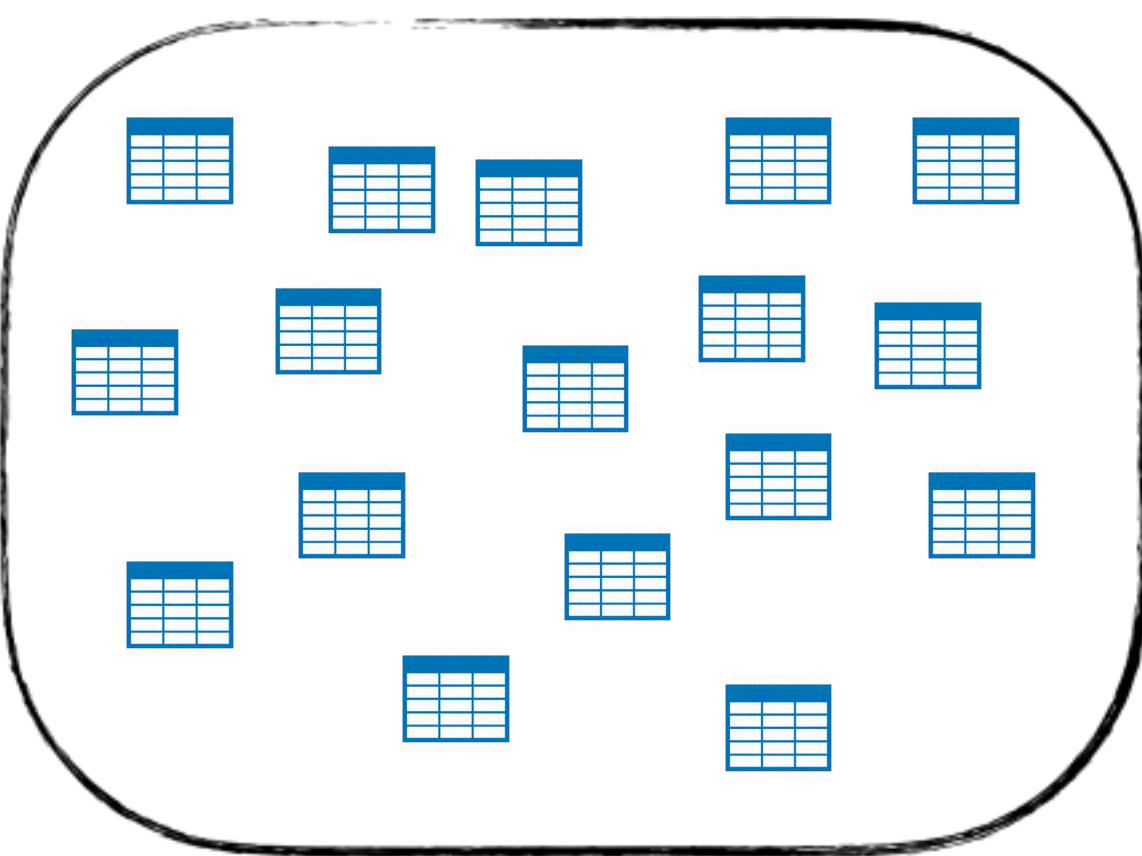


fault  
tolerance

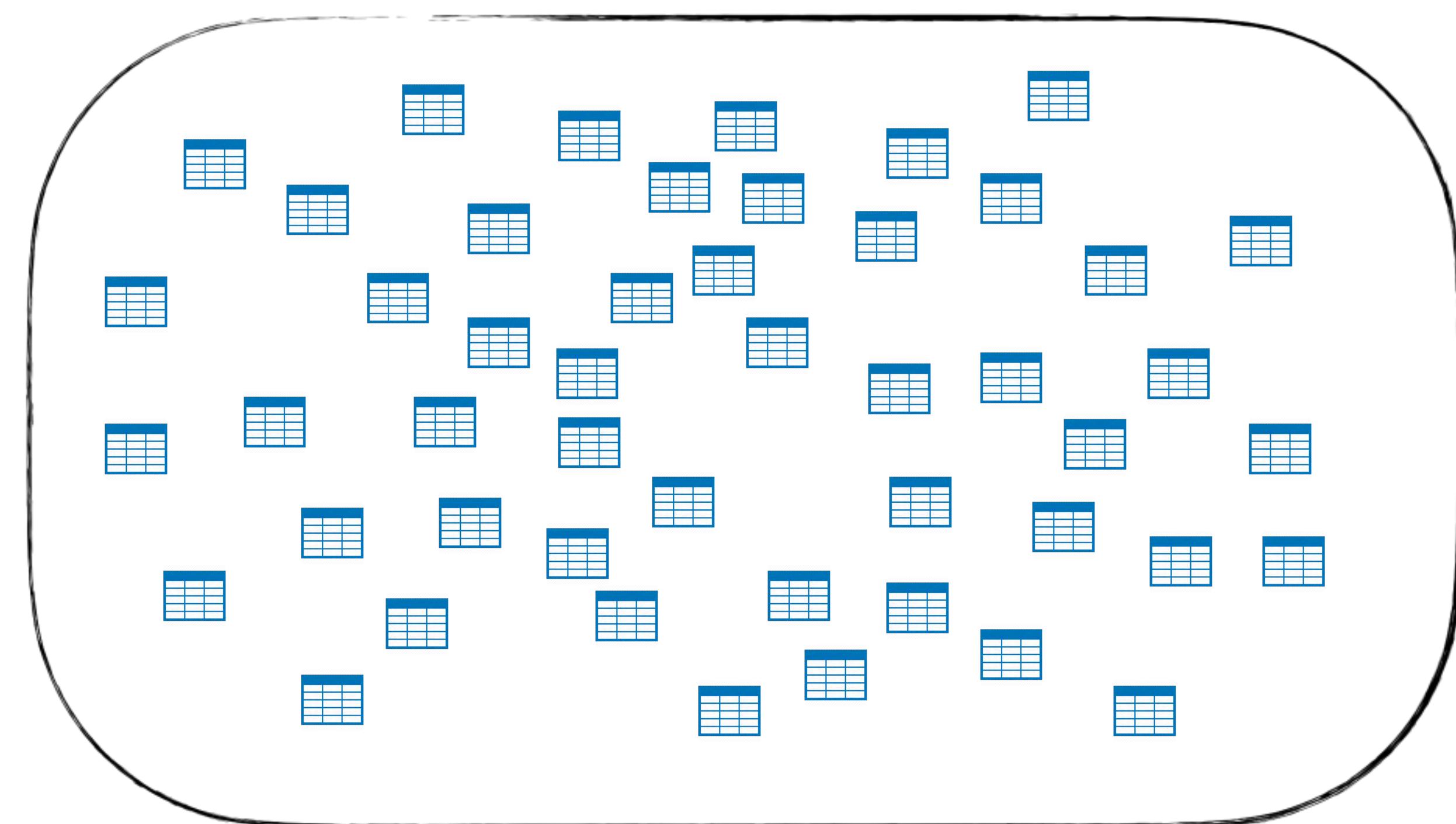
# decomposing data



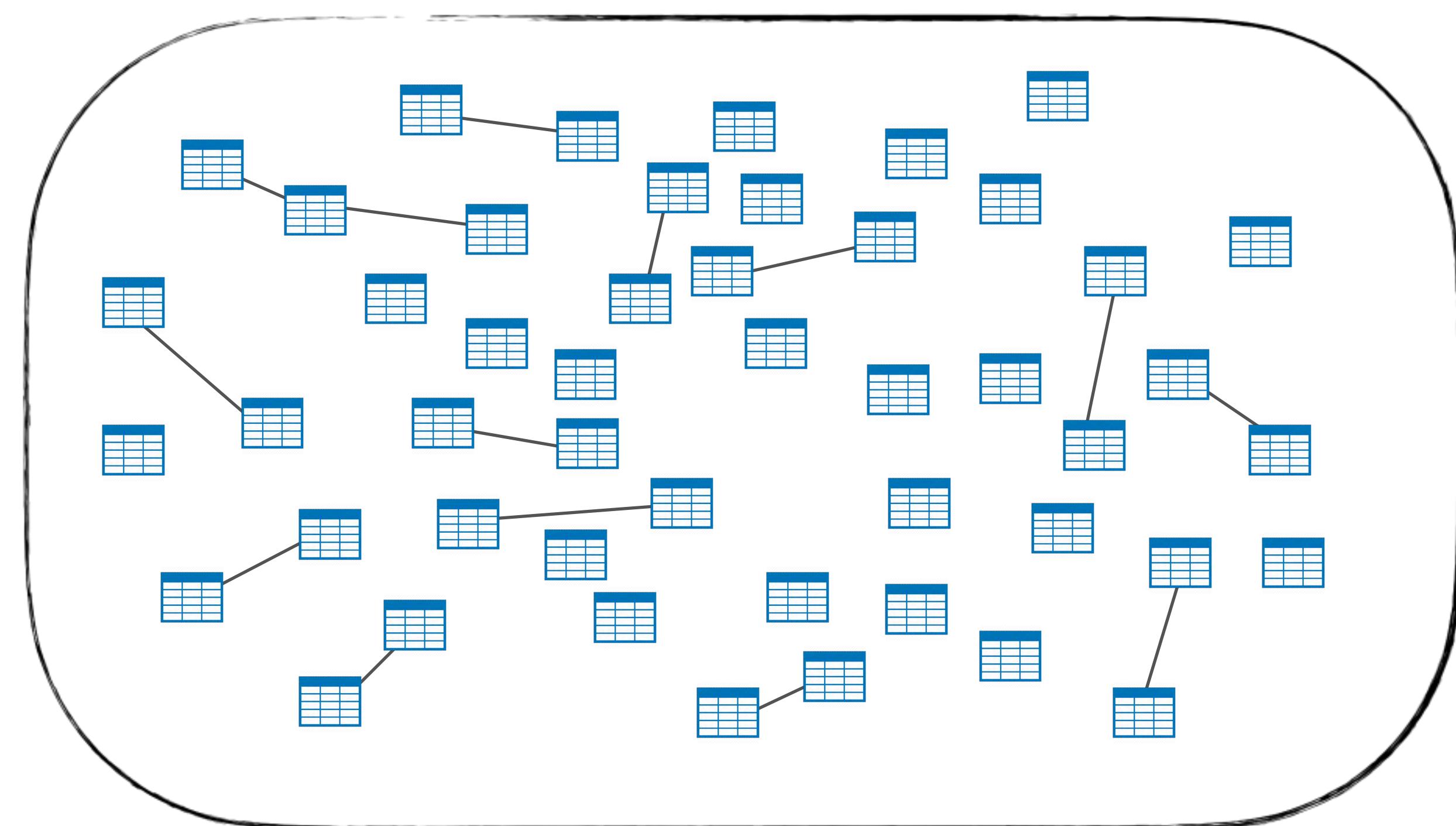
# decomposing data



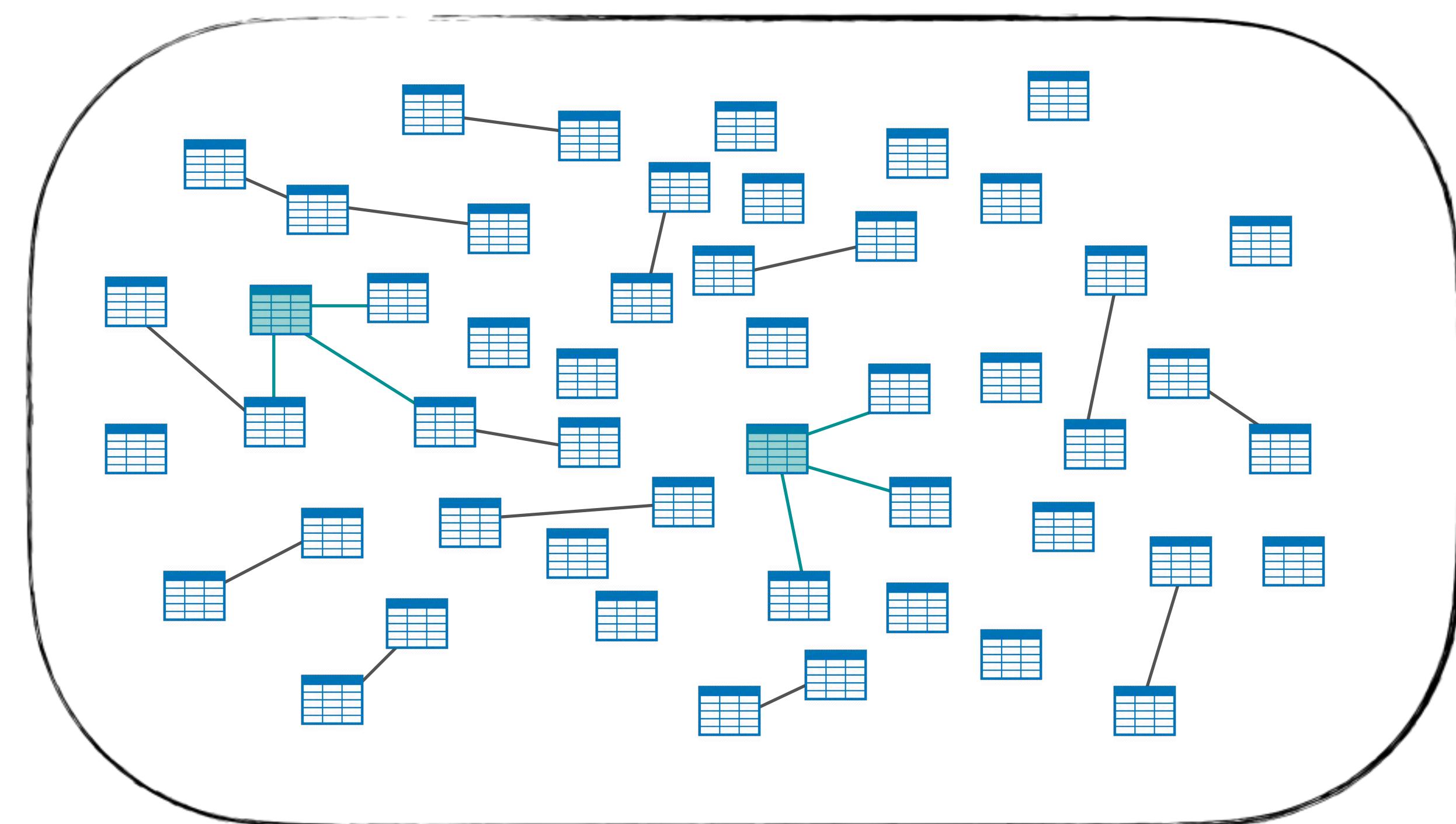
# decomposing data



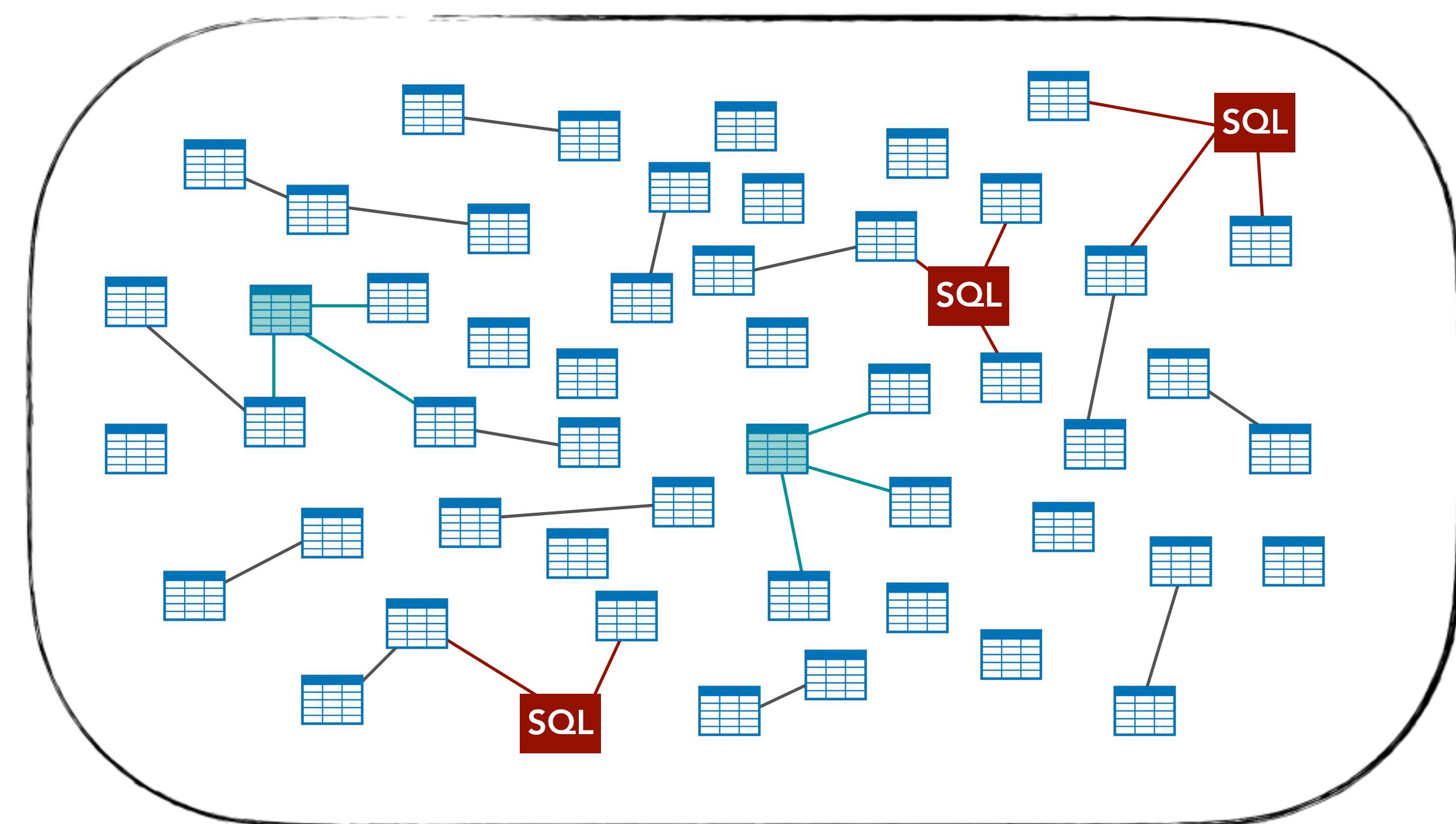
# decomposing data



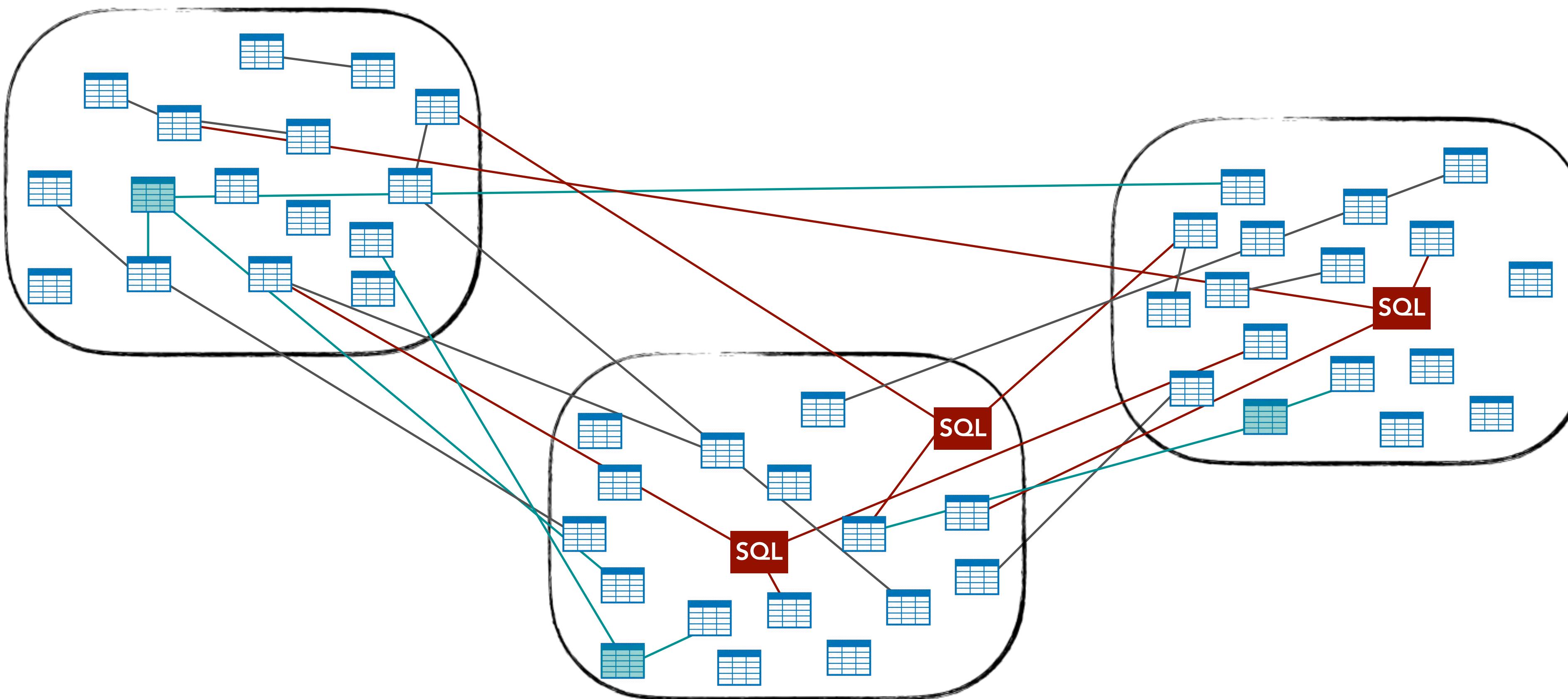
# decomposing data



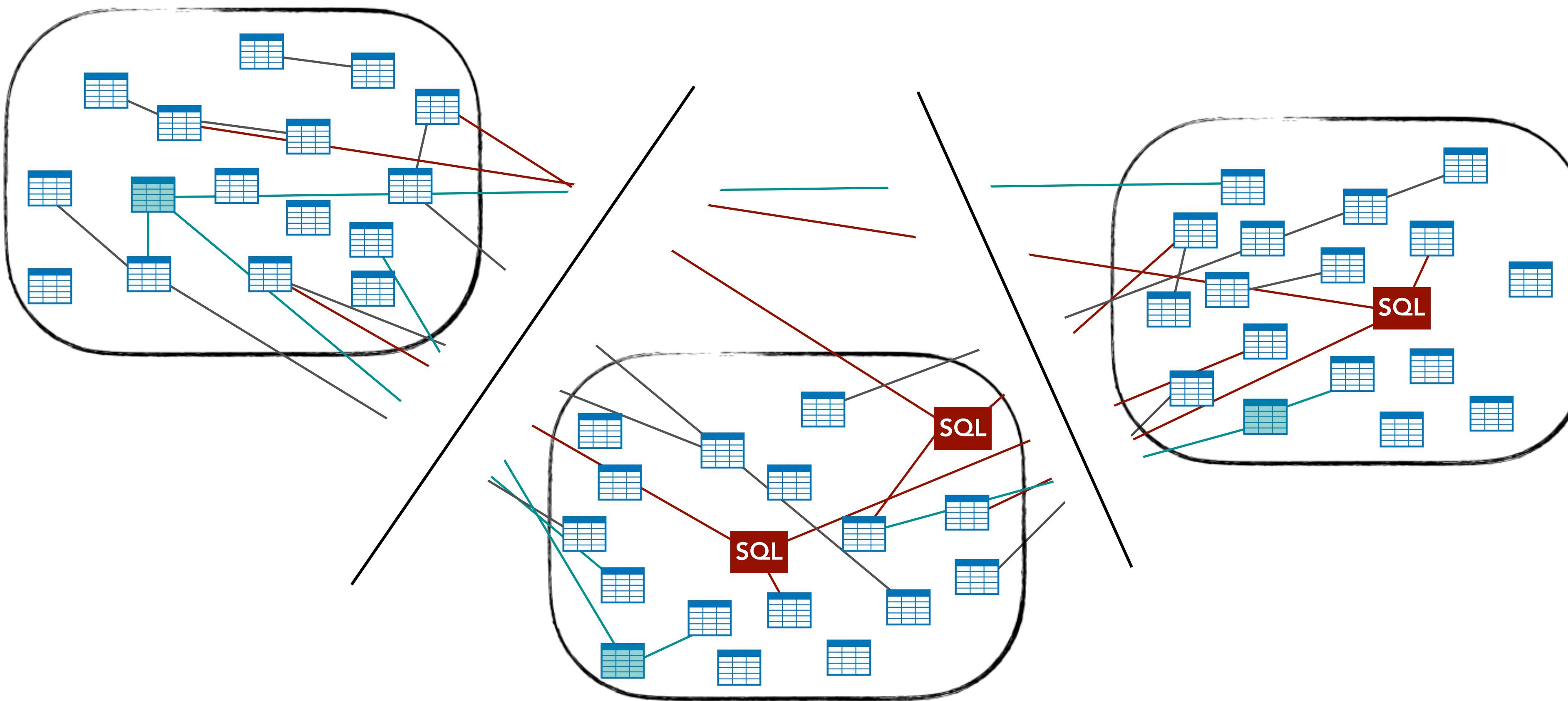
# decomposing data



# decomposing data

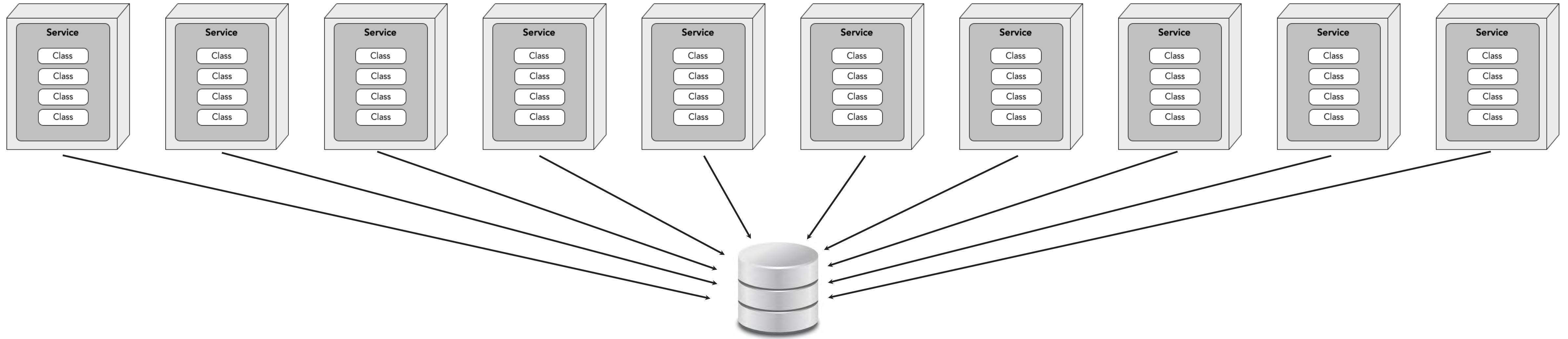


# decomposing data

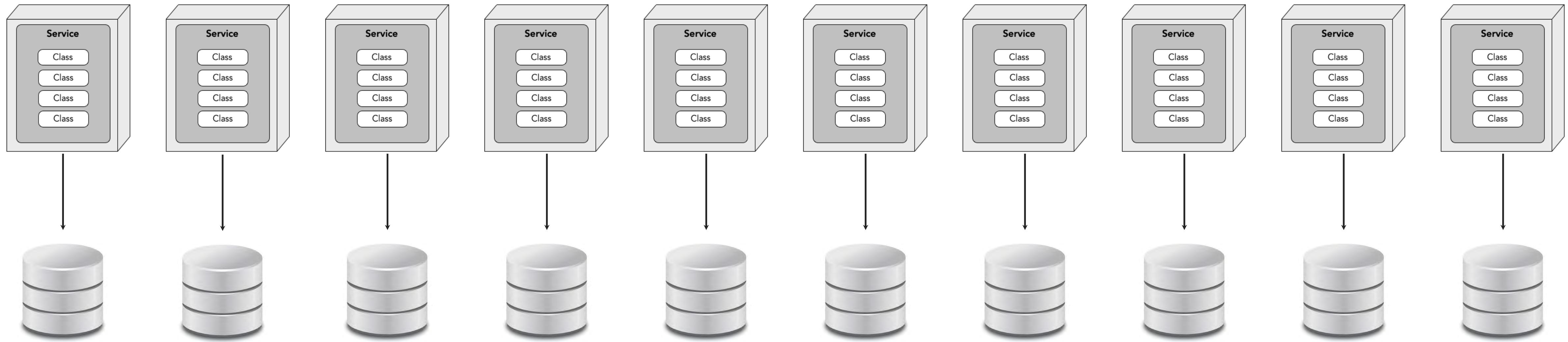


# Data Domains

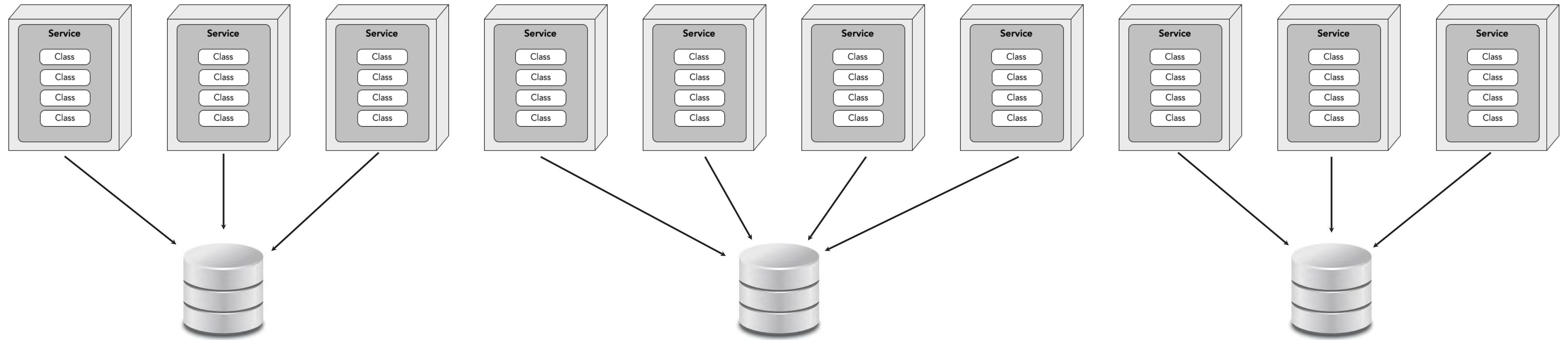
# single monolithic database



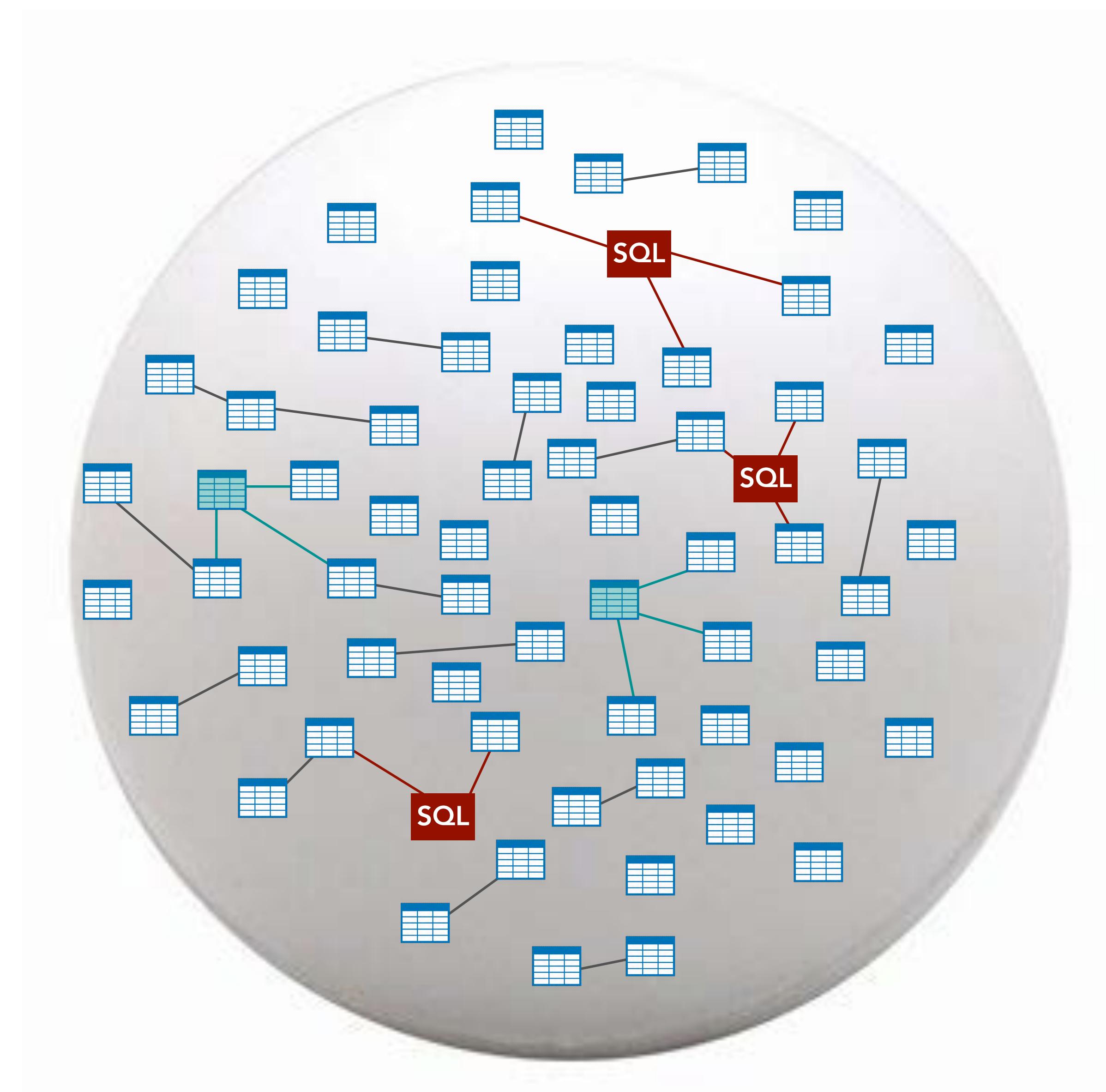
# separate databases for each service



# data domains



# data domains



# data domains



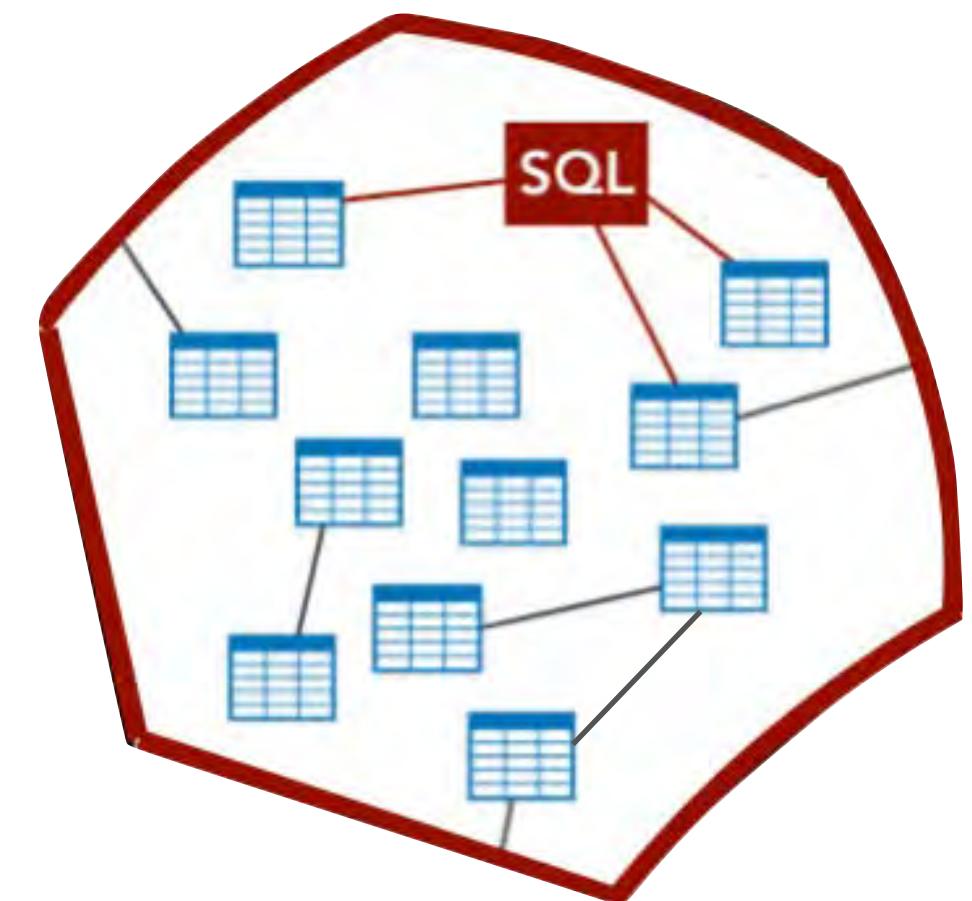
# data domains



# data domains



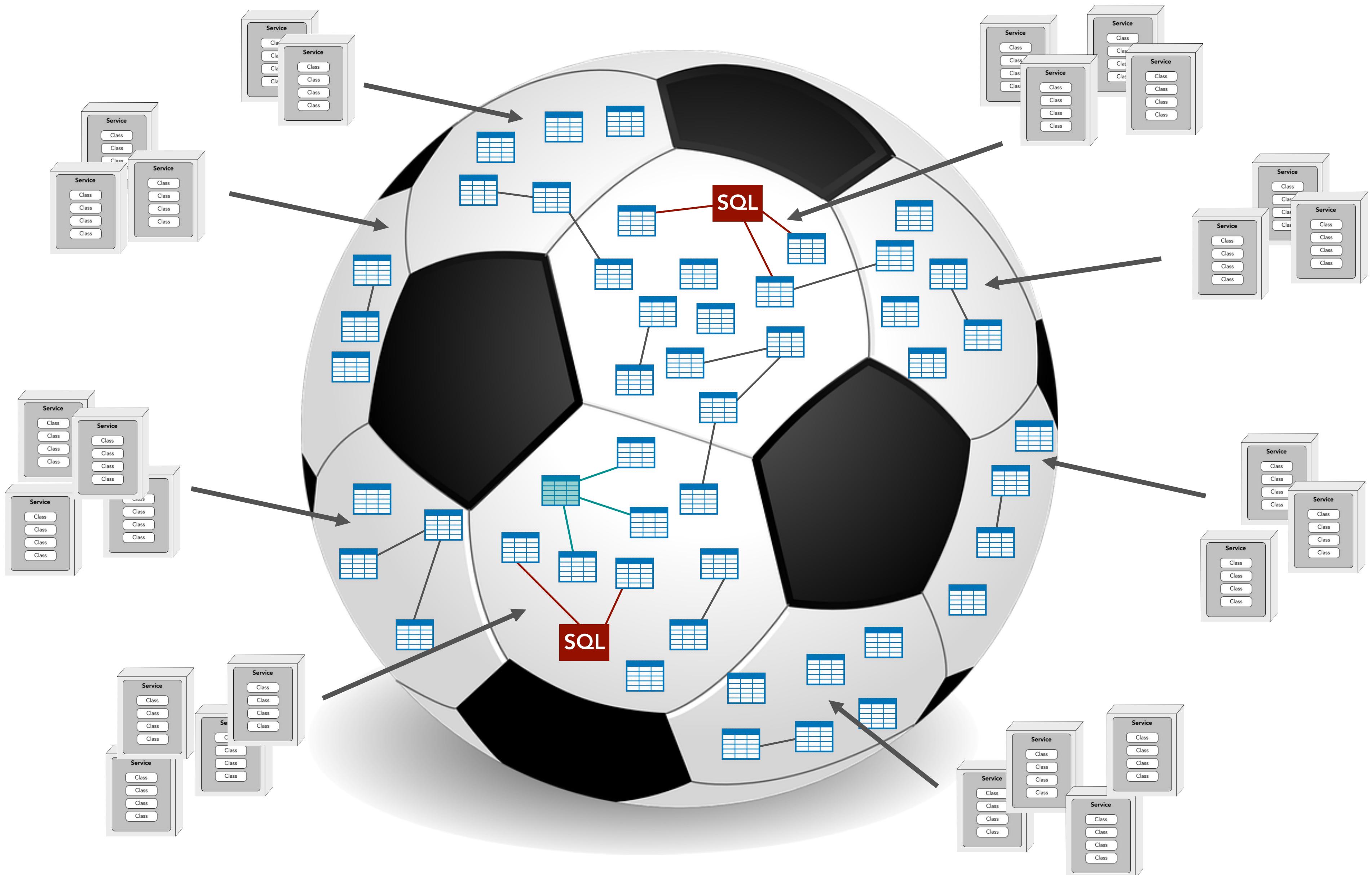
# data domains



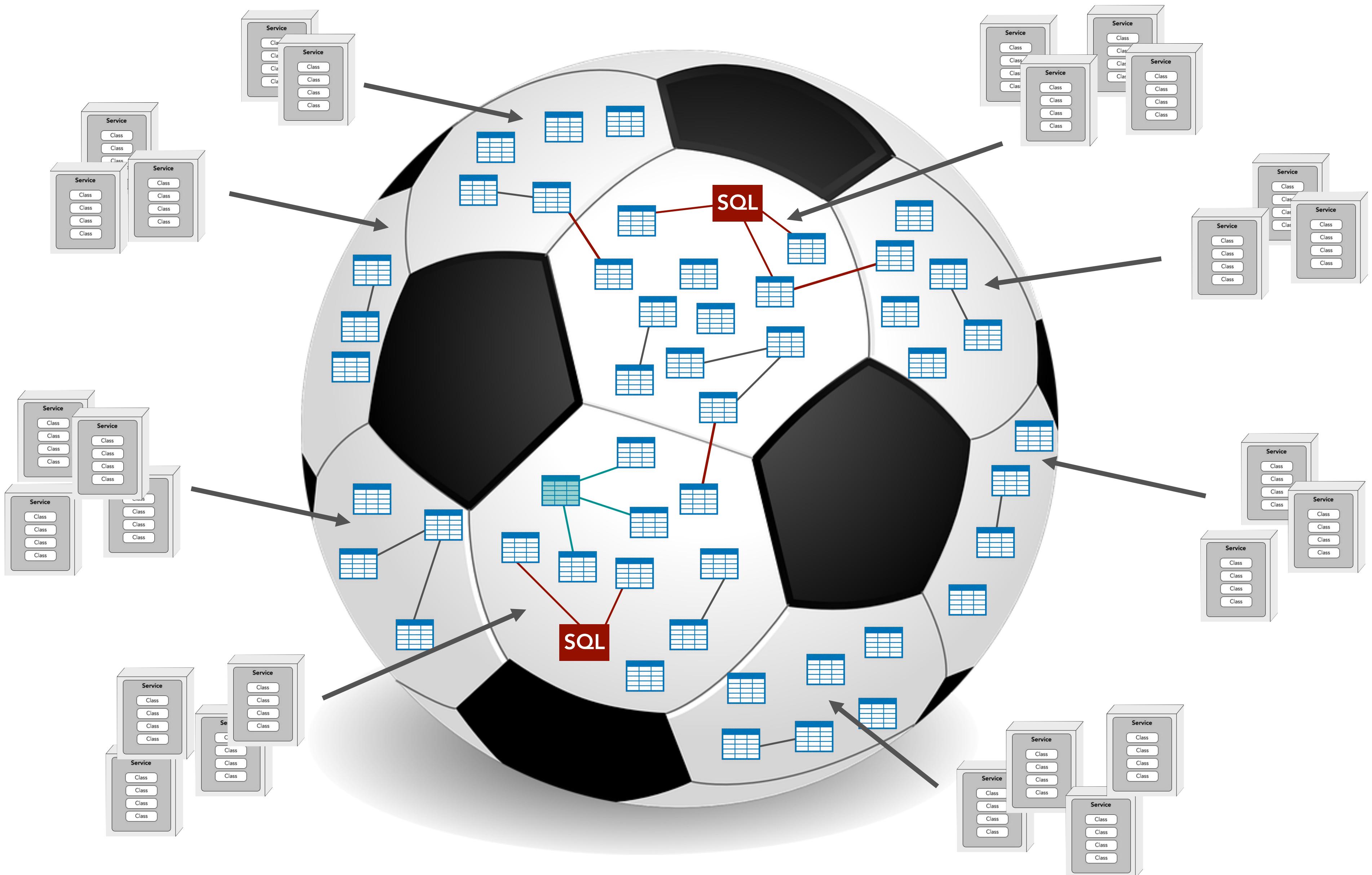
# data domains



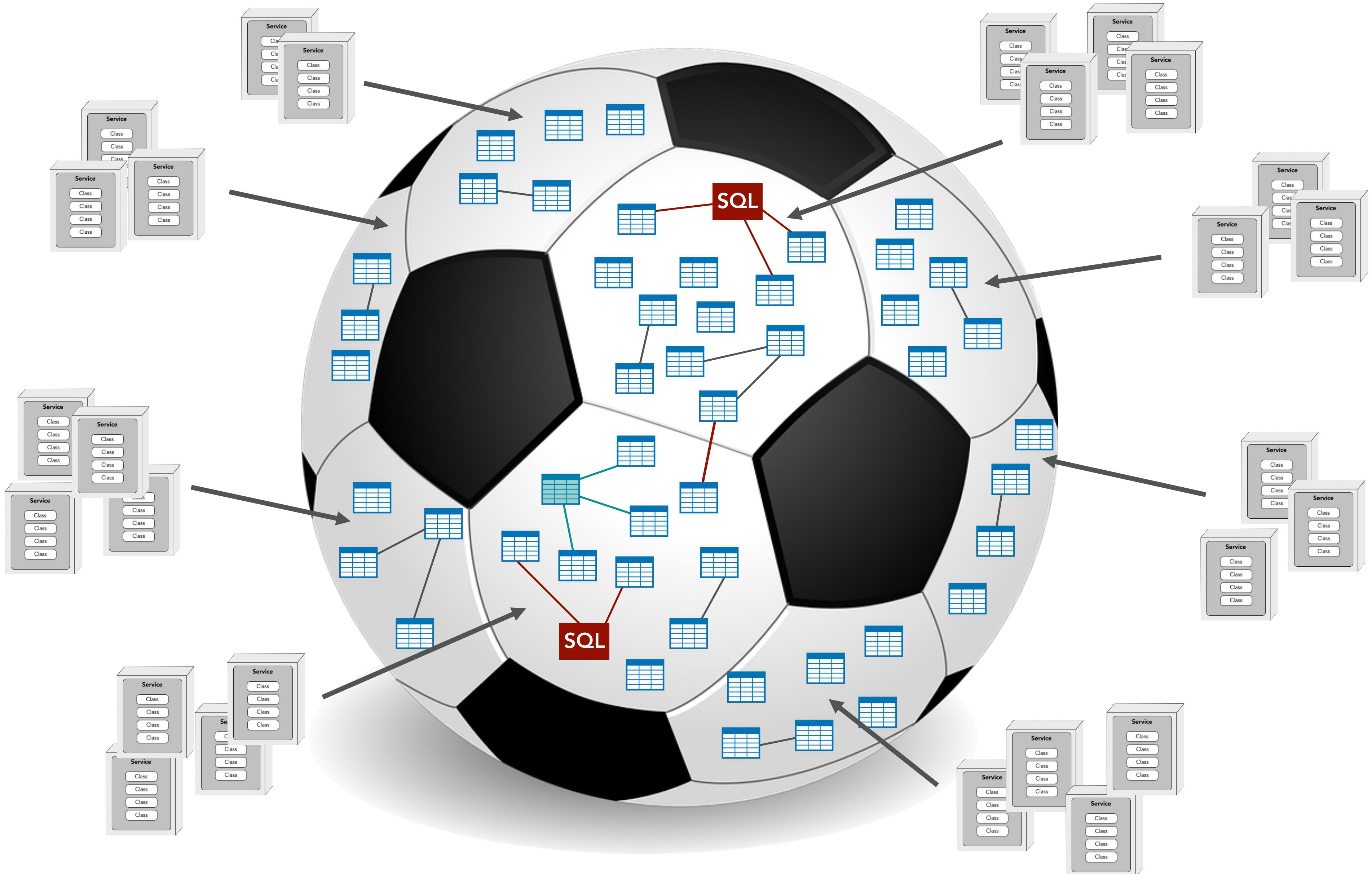
# data domains



# data domains



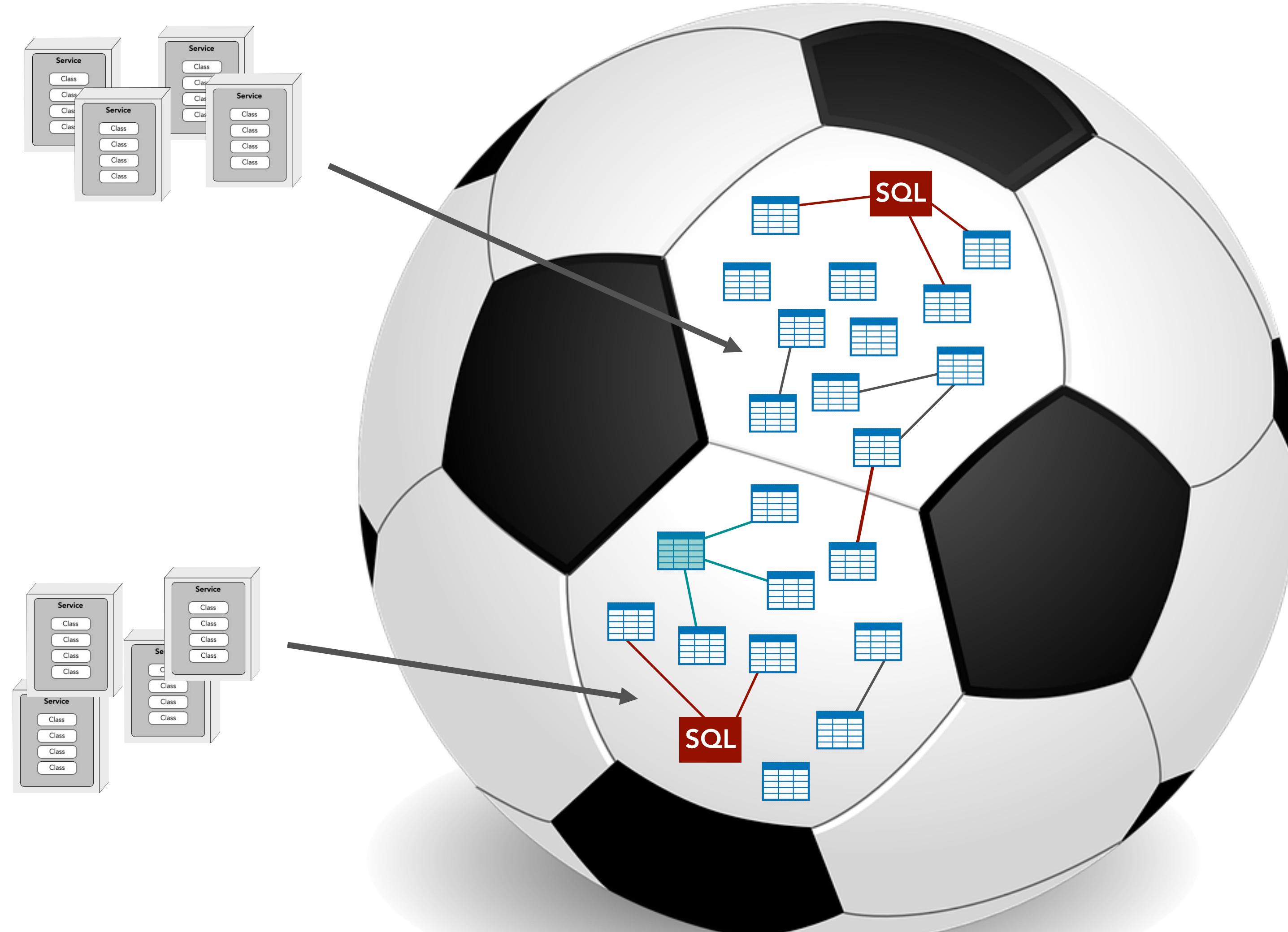
# data domains



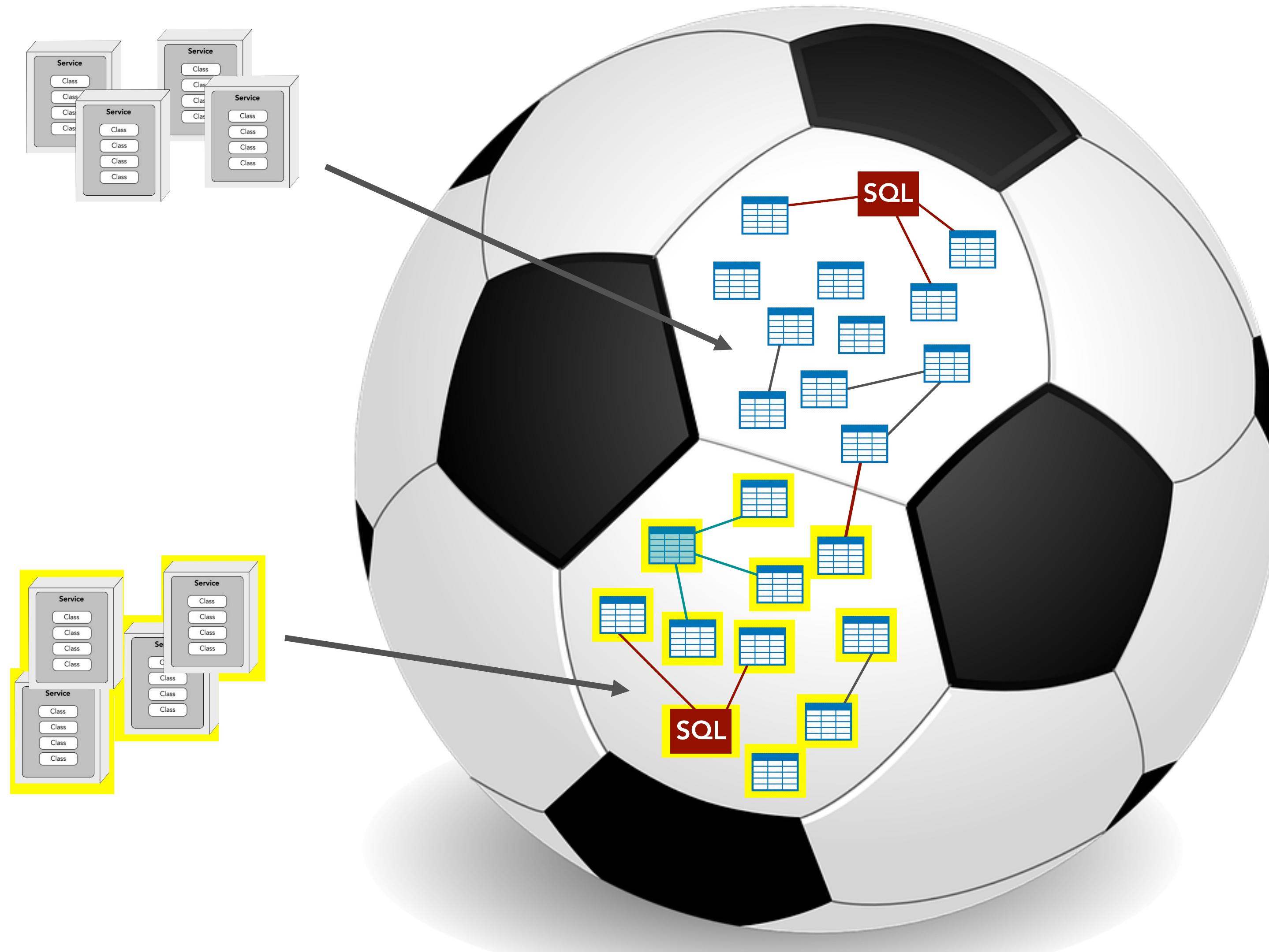
# data domains



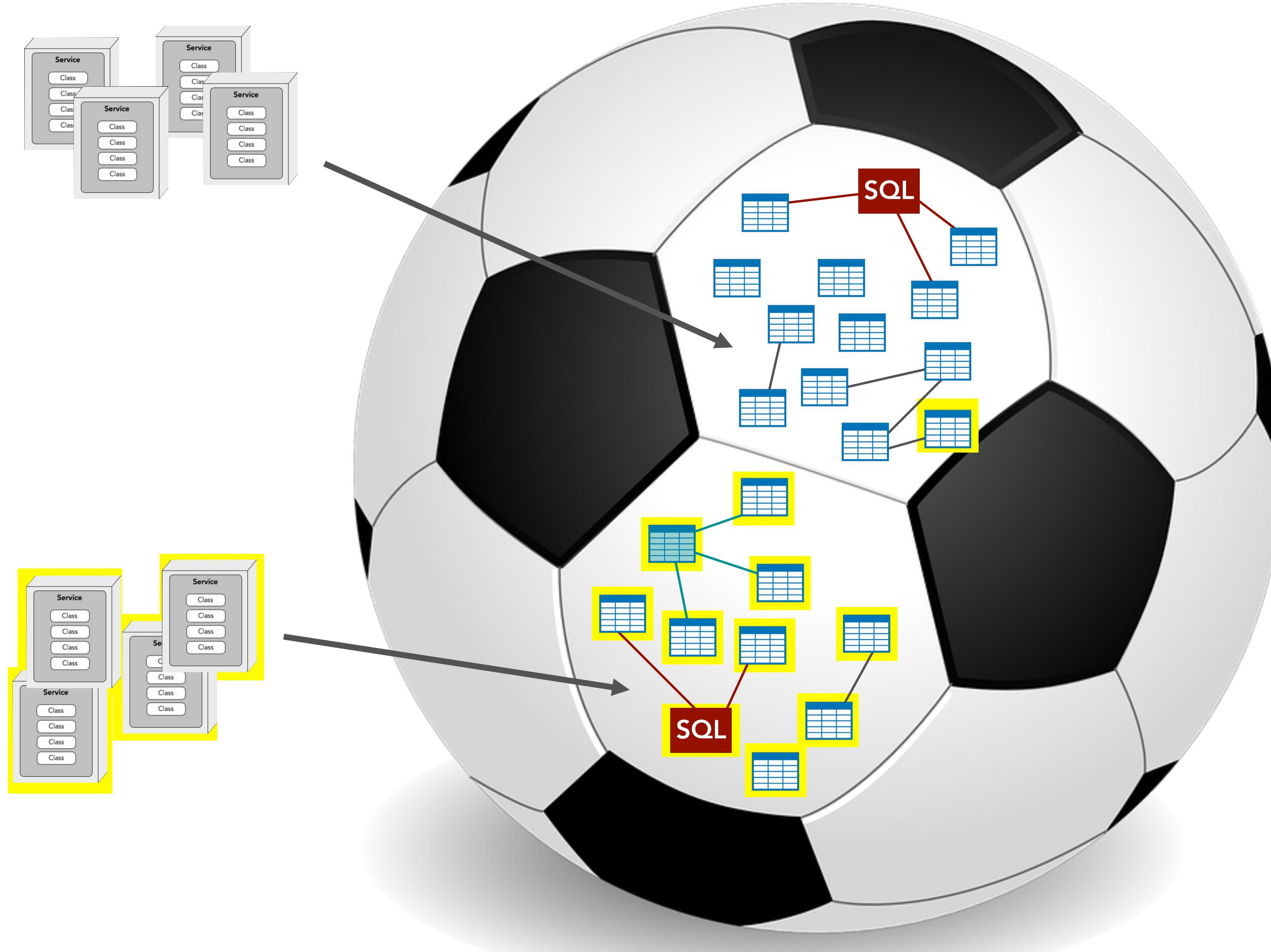
# data domains



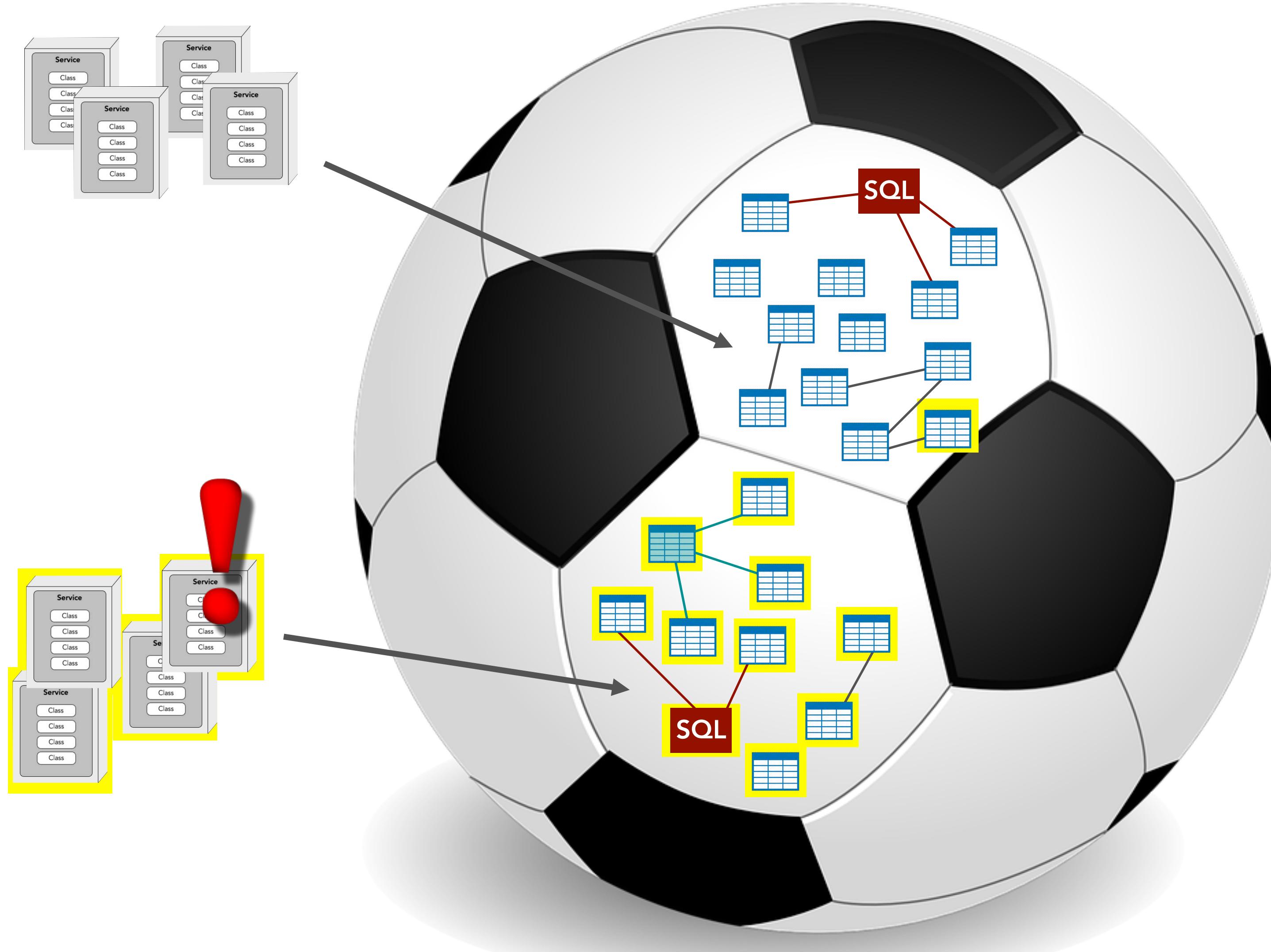
# data domains



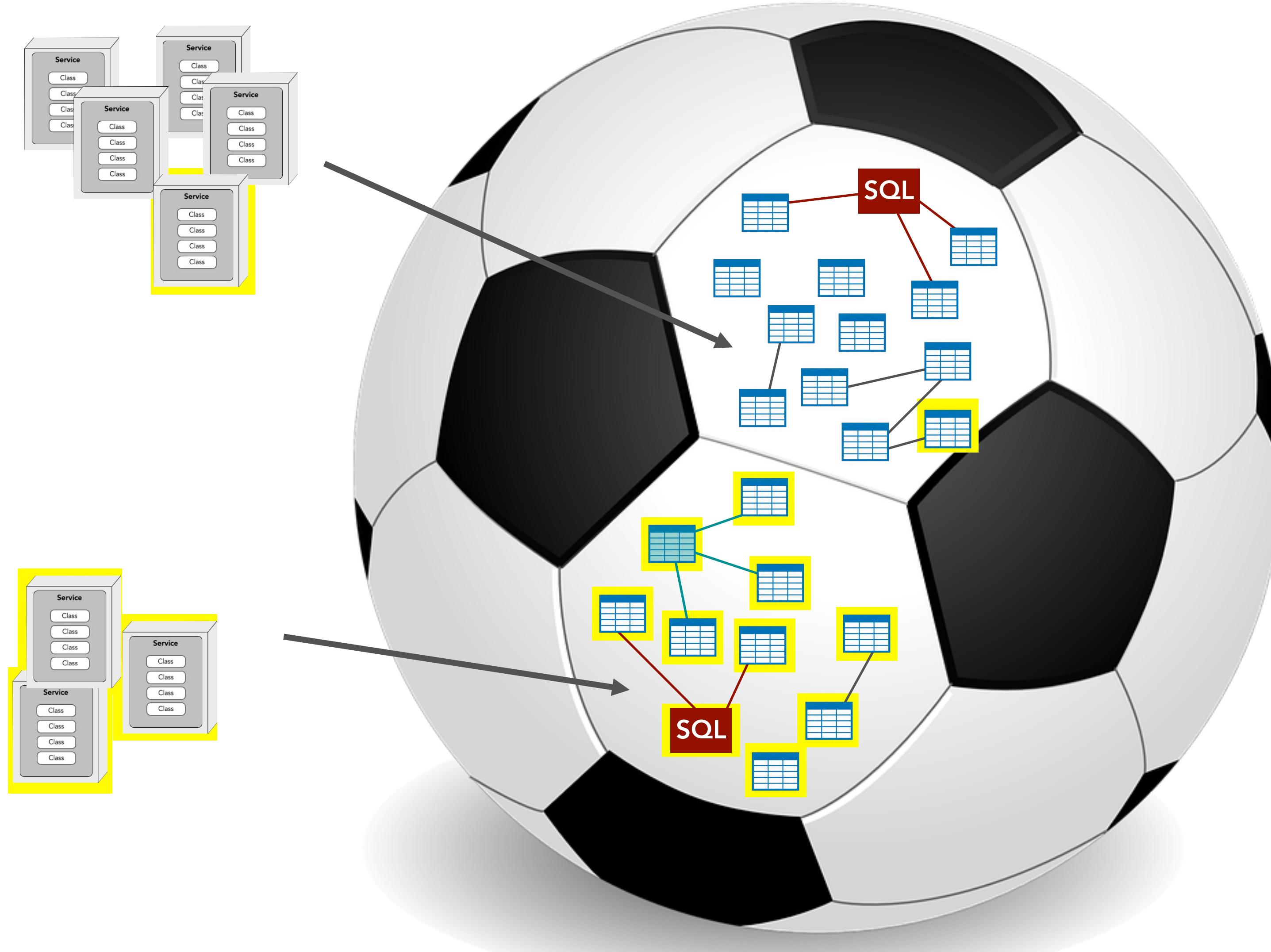
# data domains



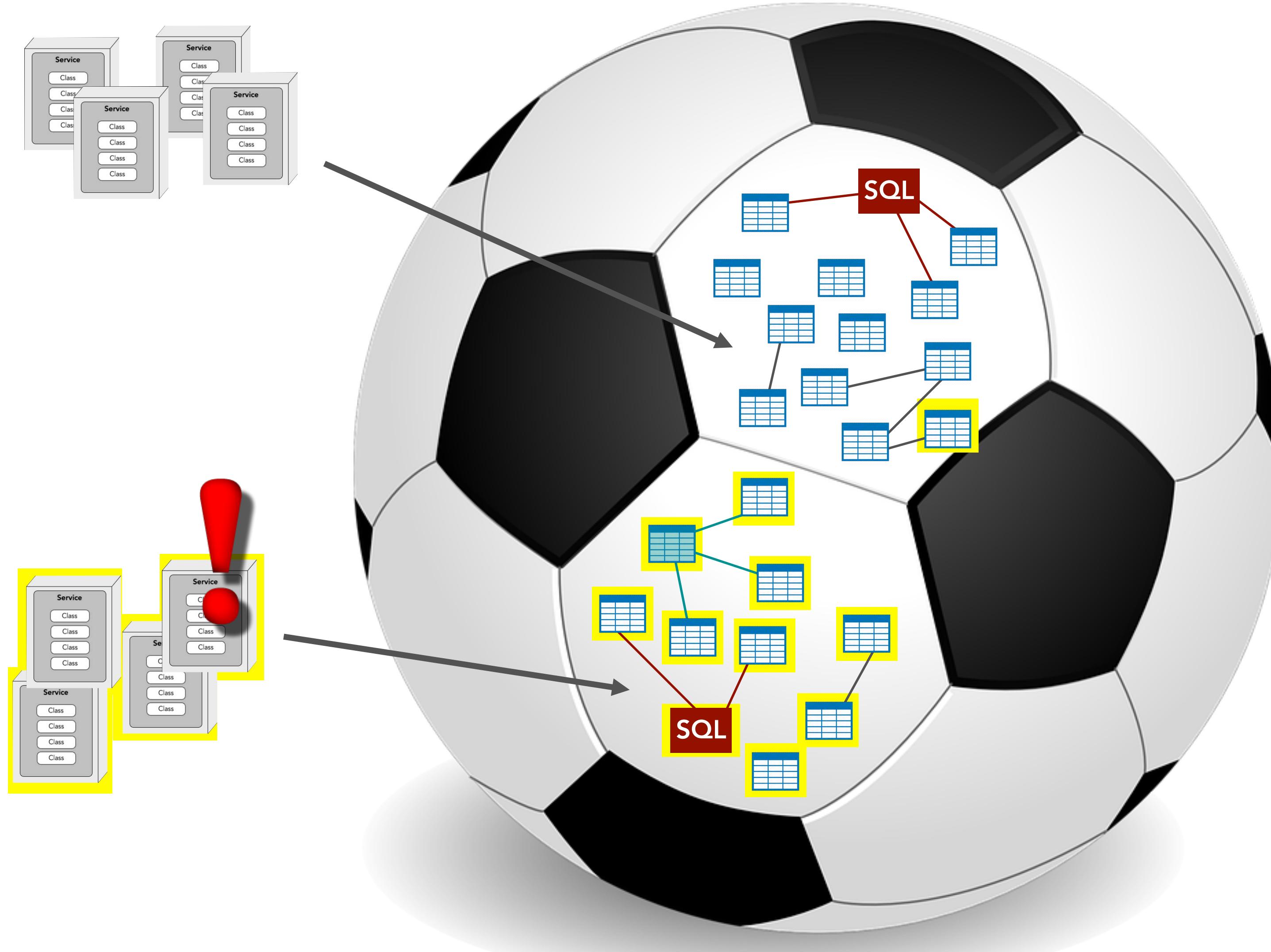
# data domains



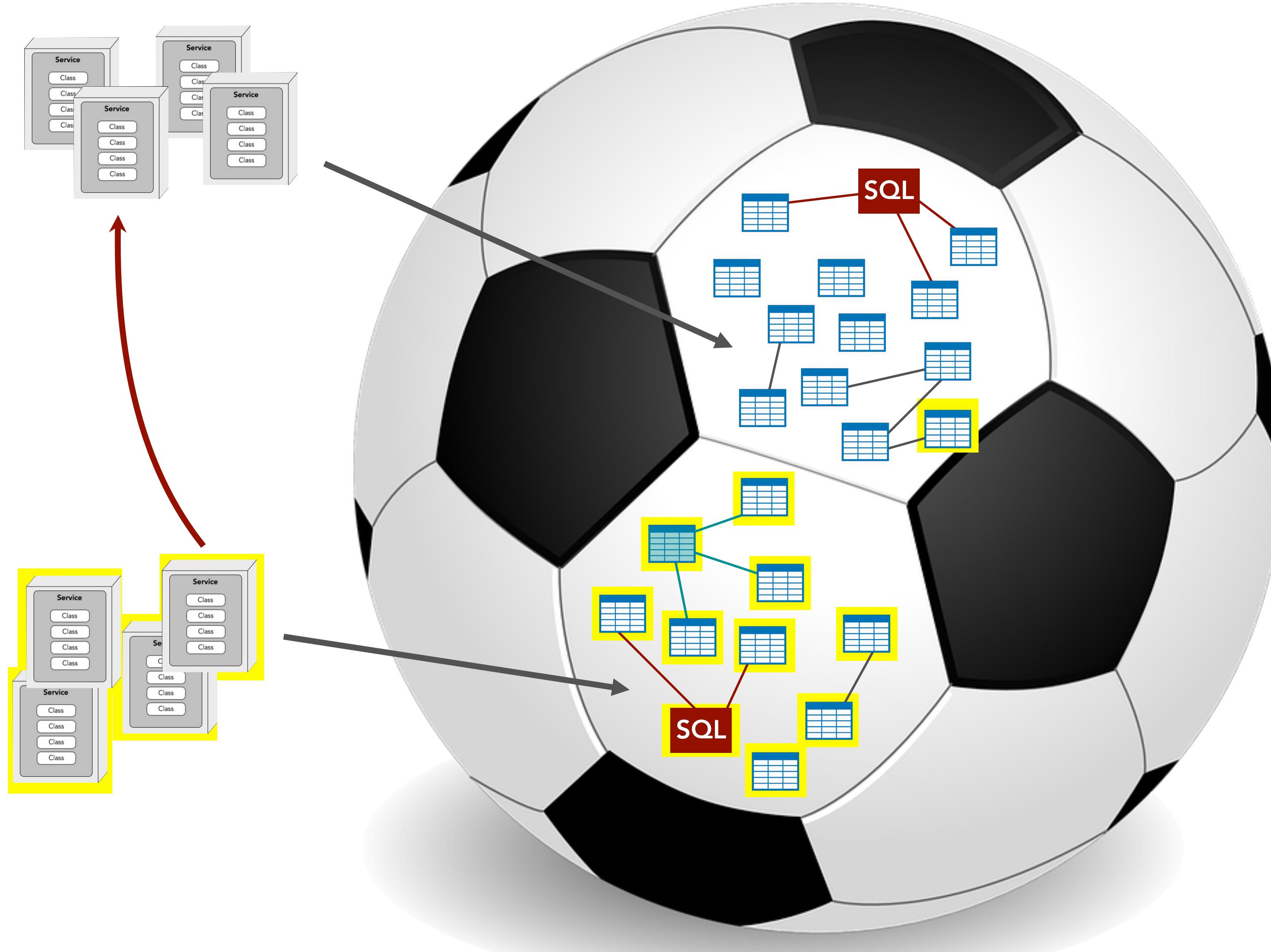
# data domains



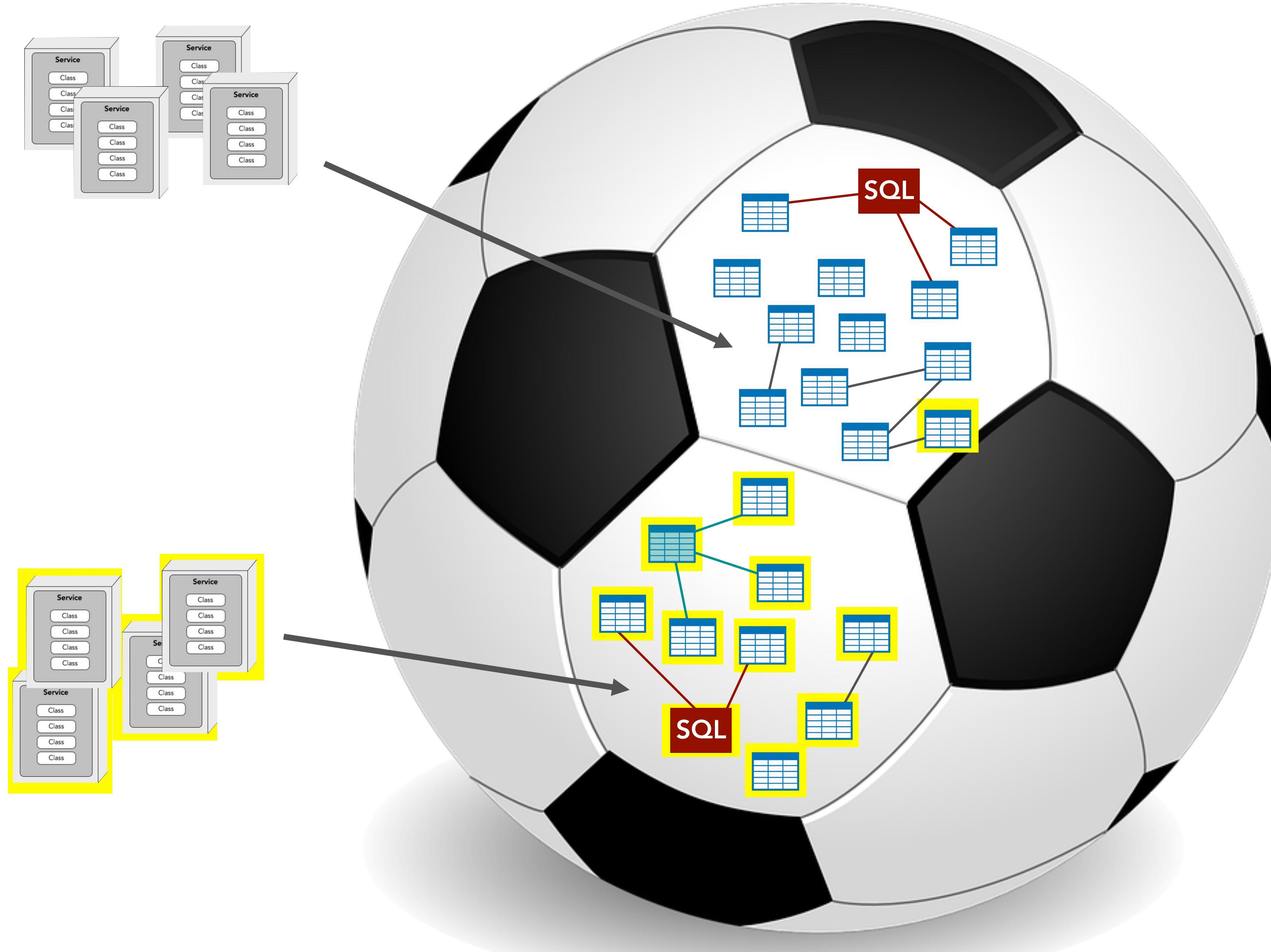
# data domains



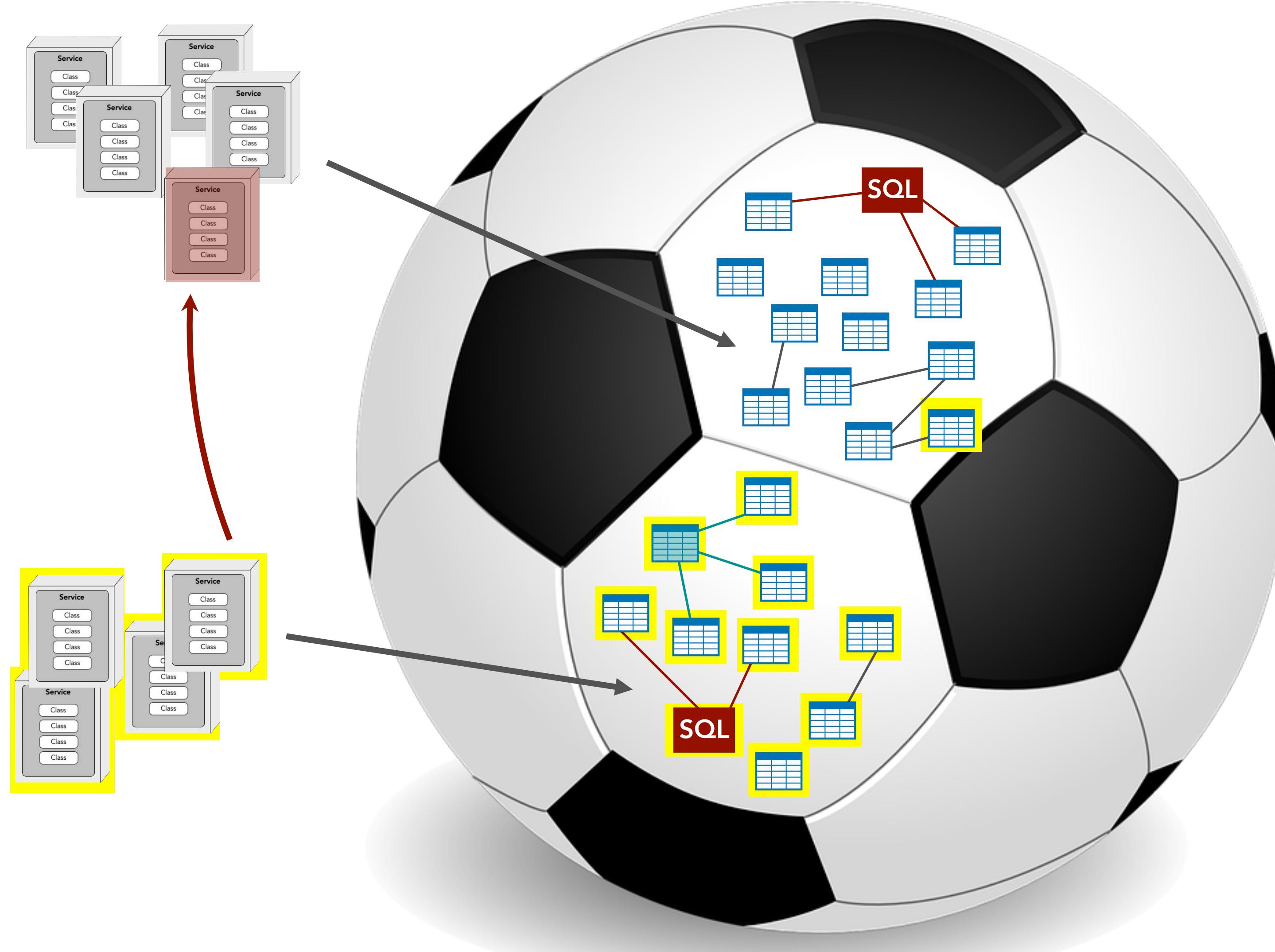
# data domains



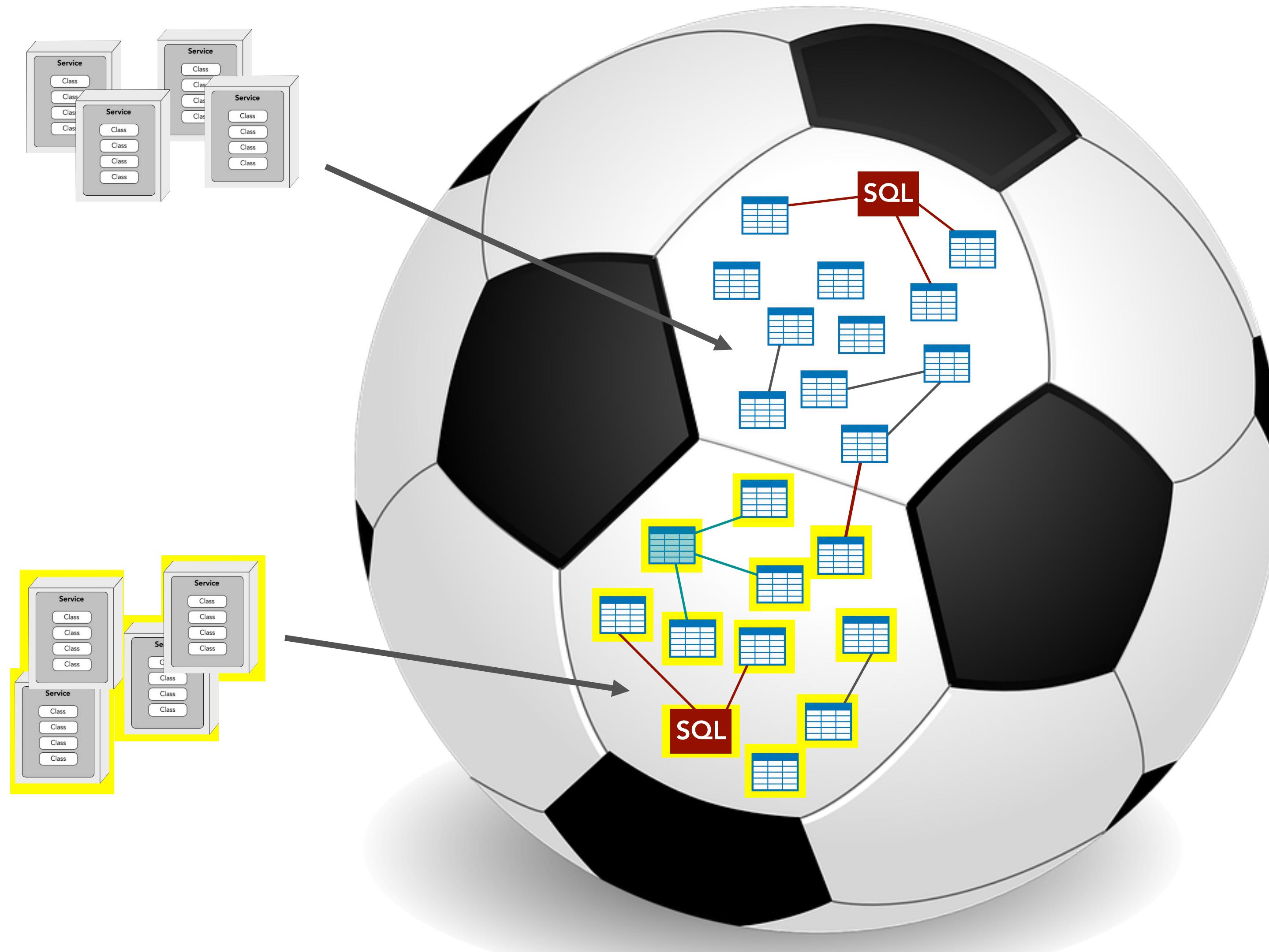
# data domains



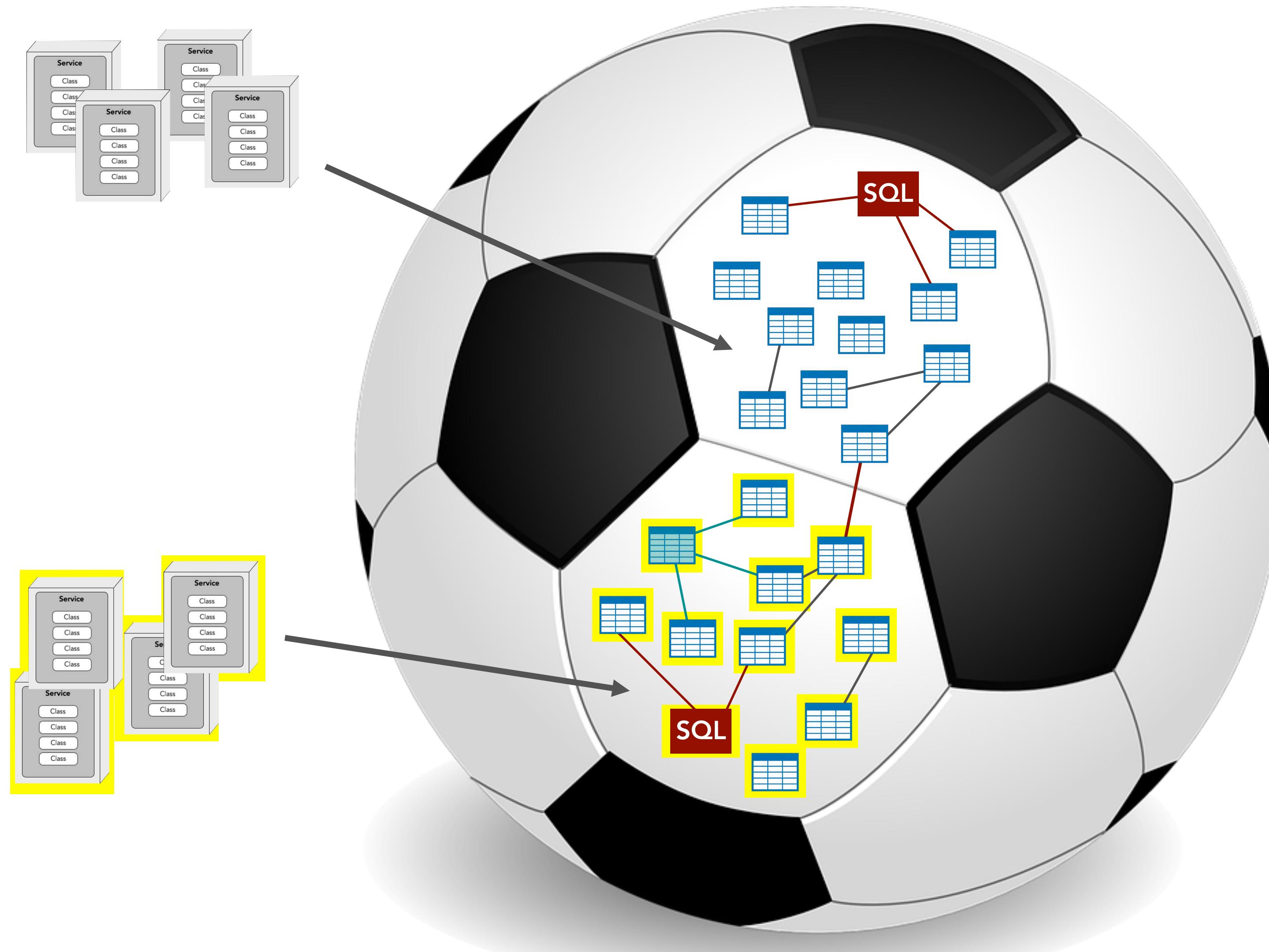
# data domains



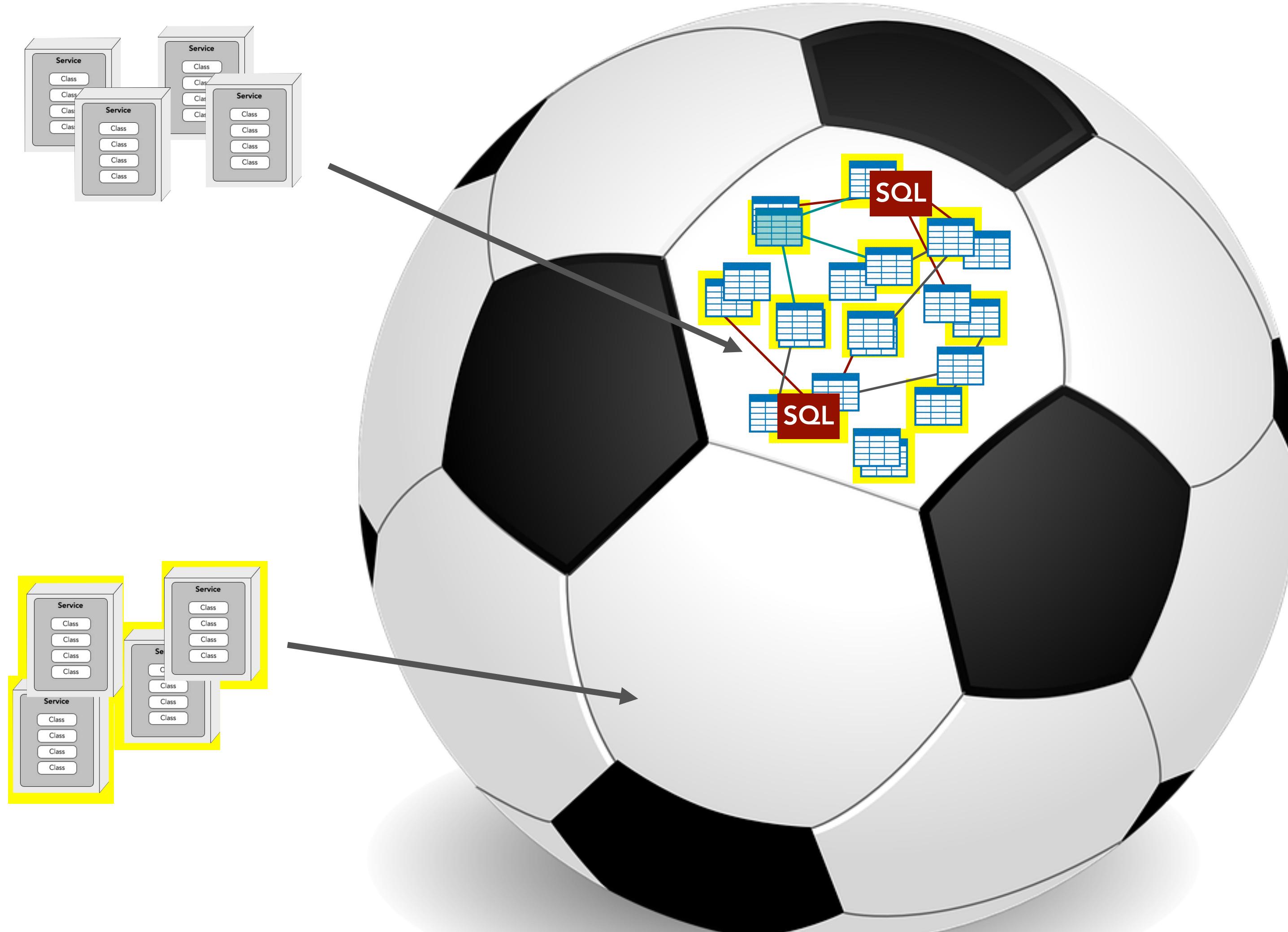
# data domains



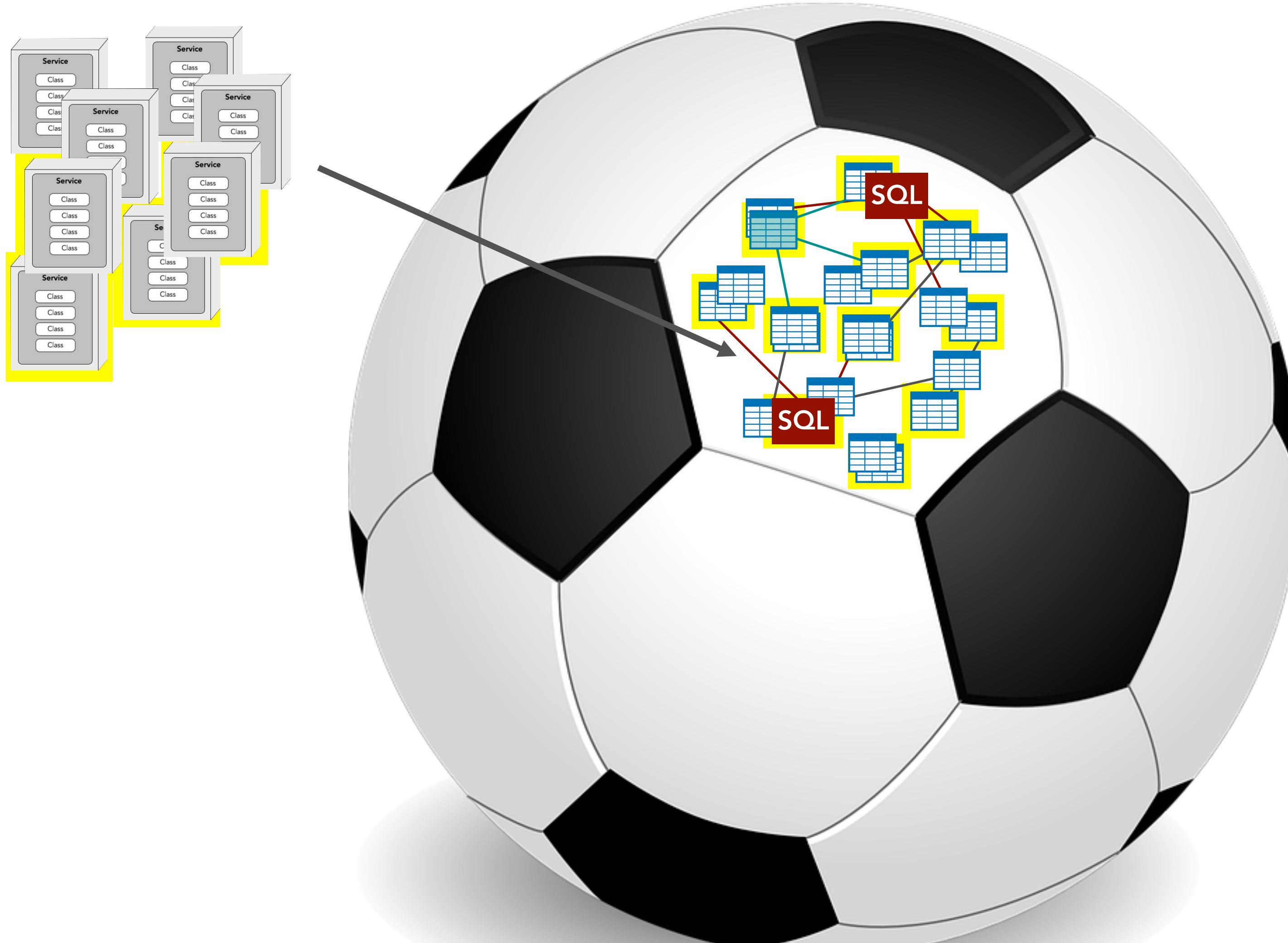
# data domains



# data domains



# data domains

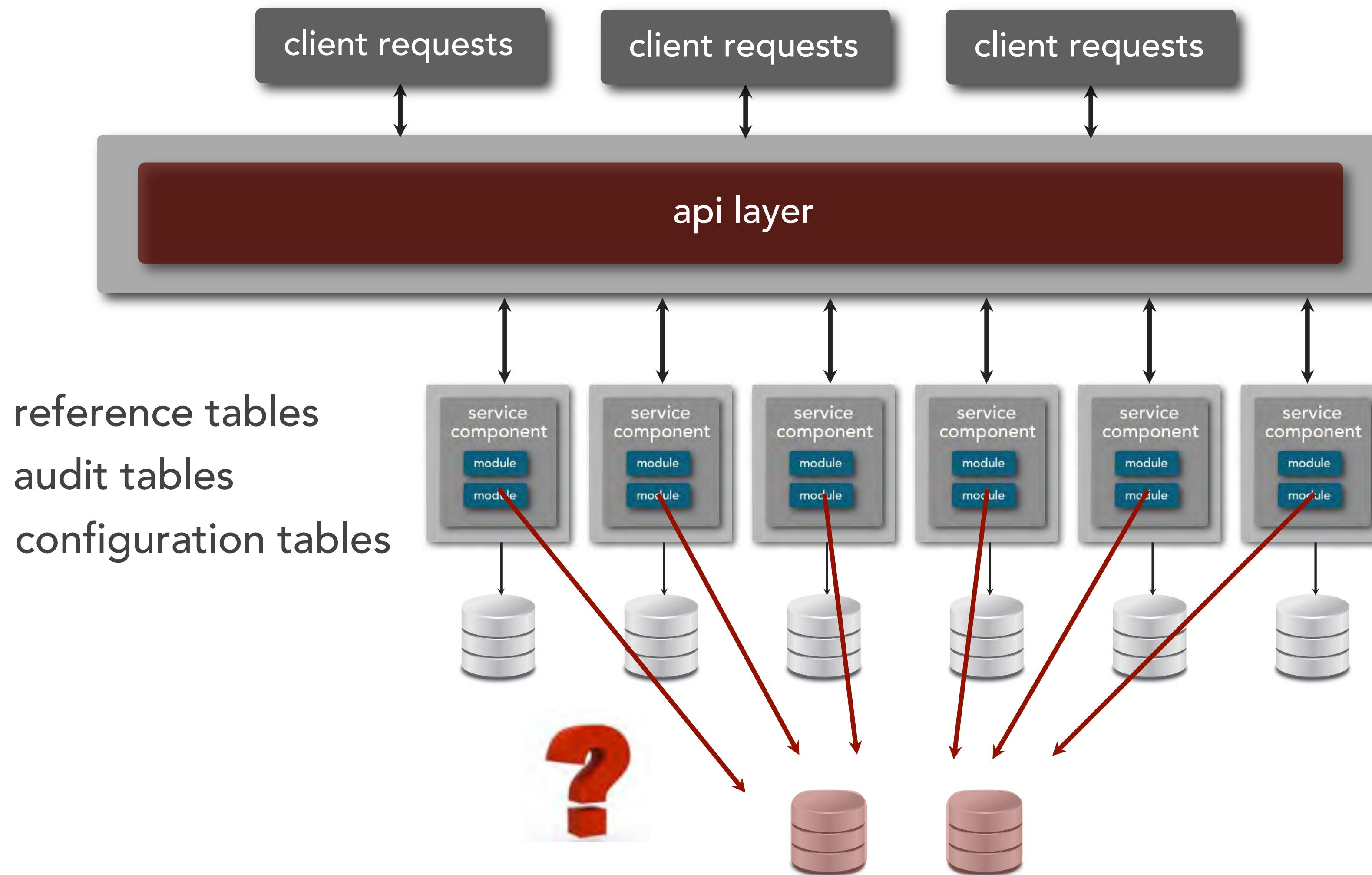


# data domains

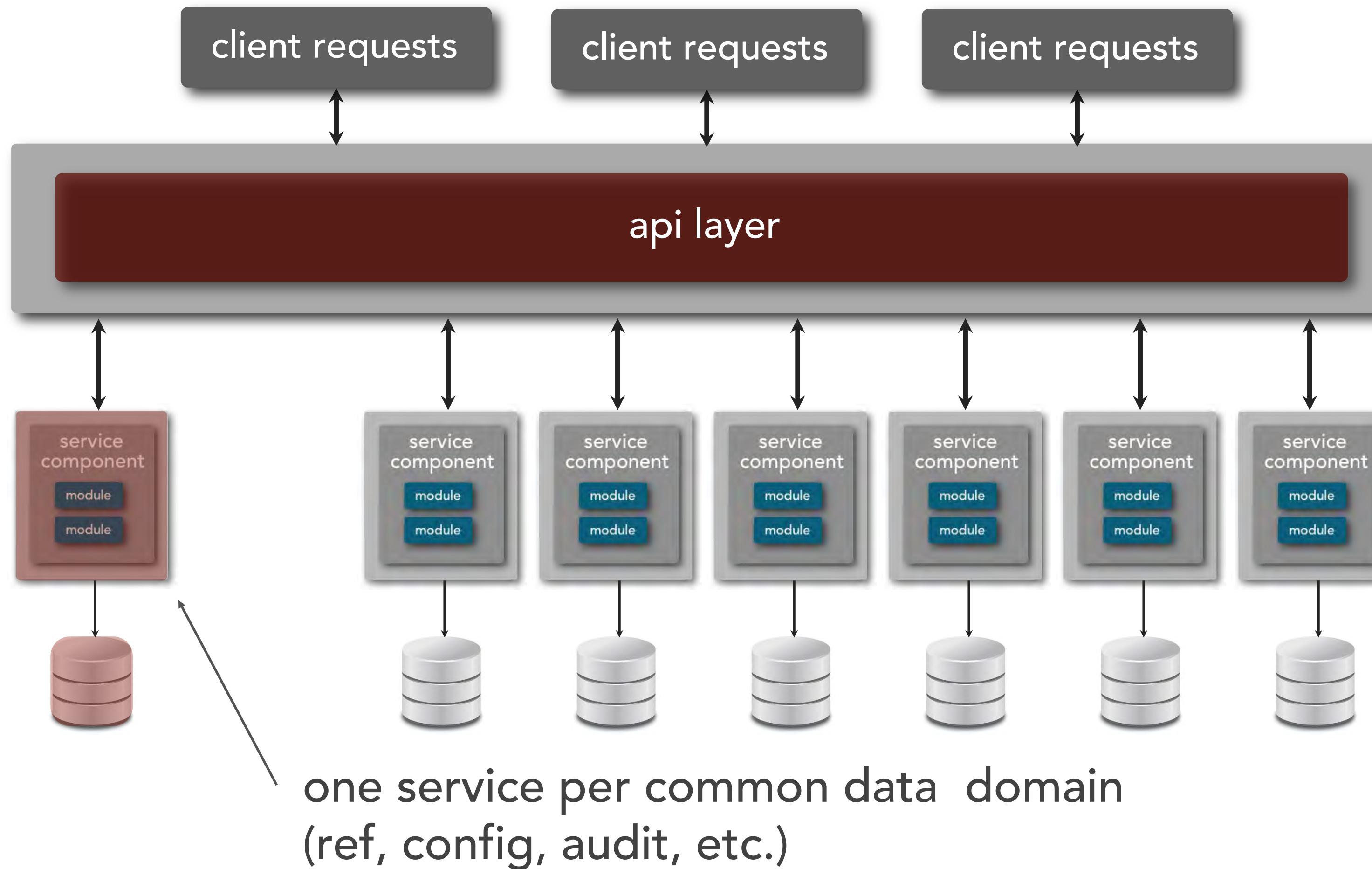


# Managing Common Data

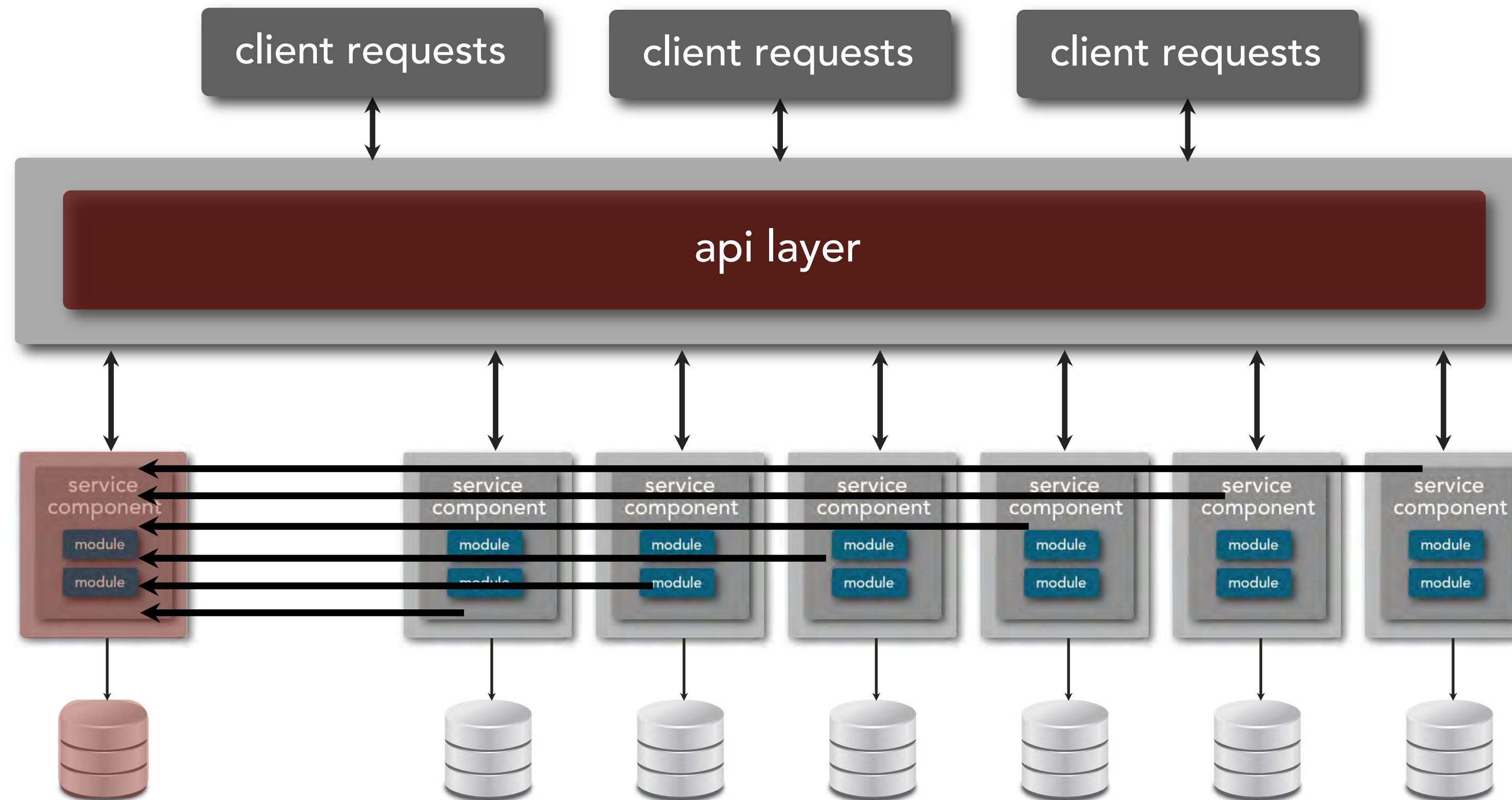
# managing common data



# managing common data

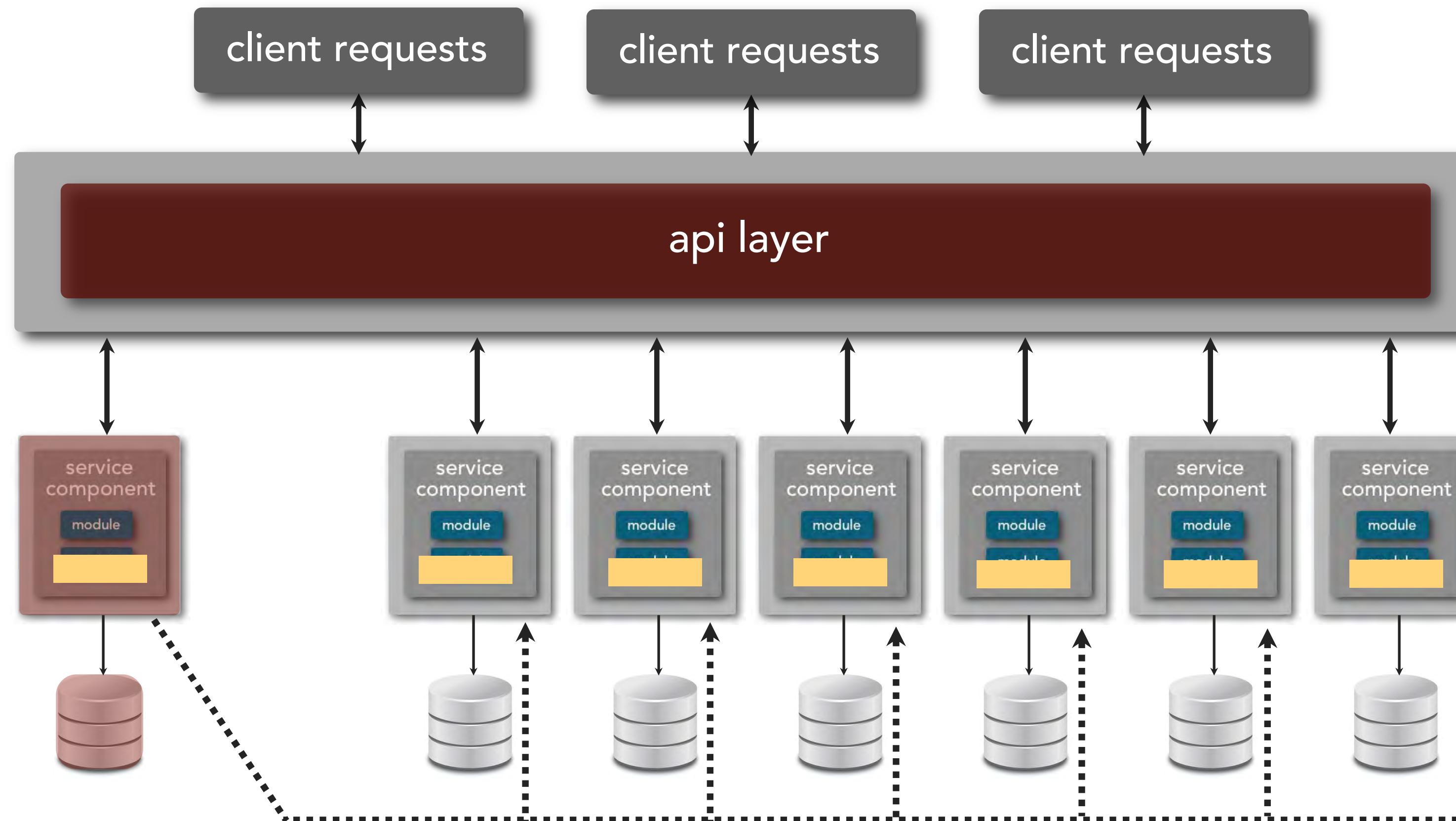


# managing common data



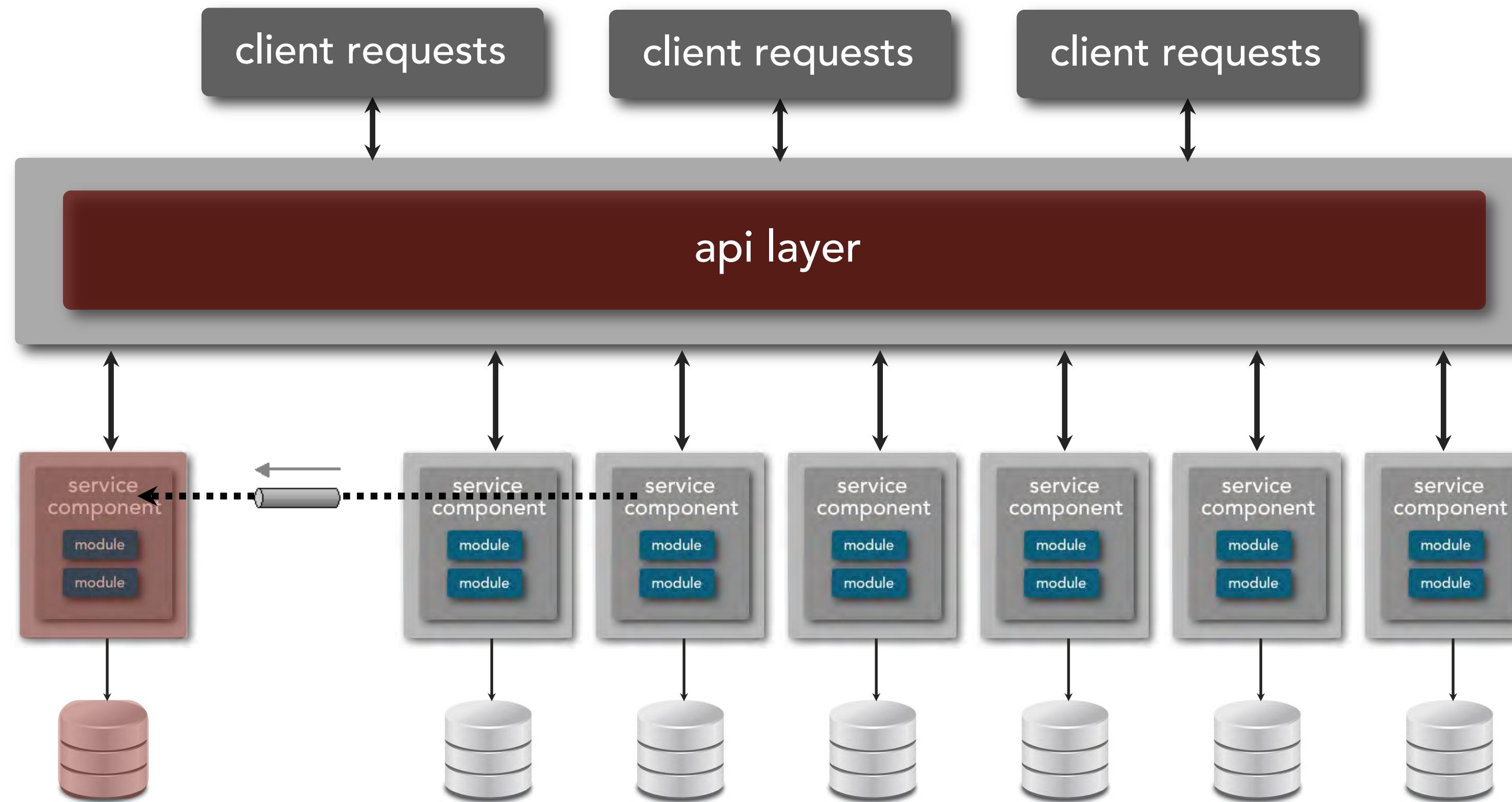
inter-service communication for reads

# managing common data



replicated cache for reads

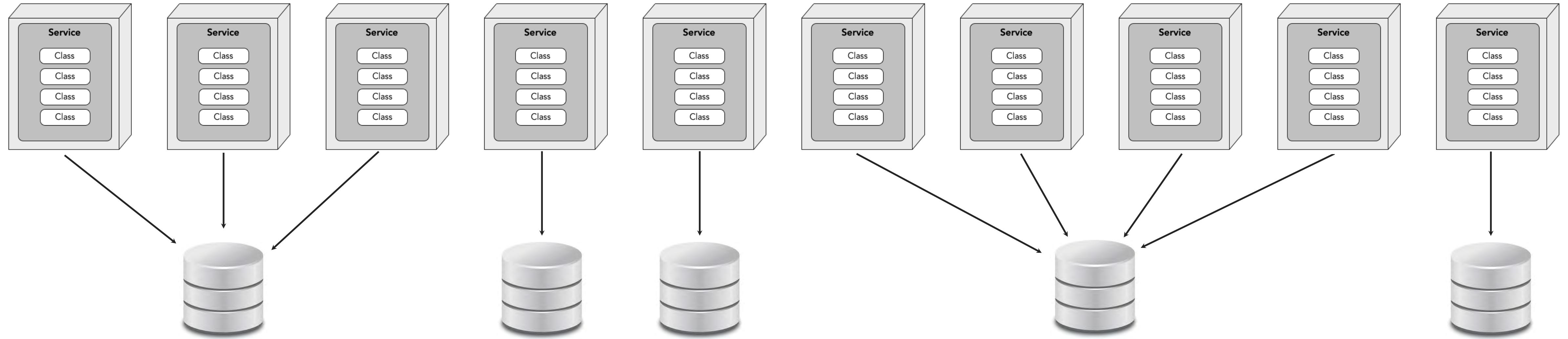
# managing common data



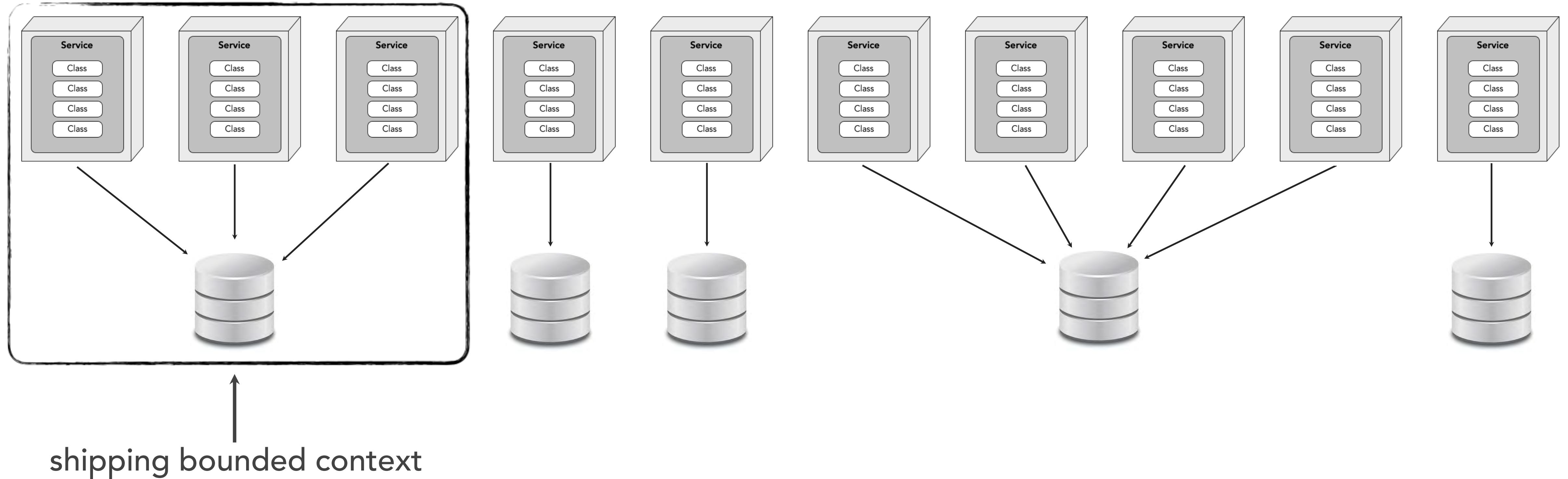
async messaging for writes

# Managing a Broader Bounded Context

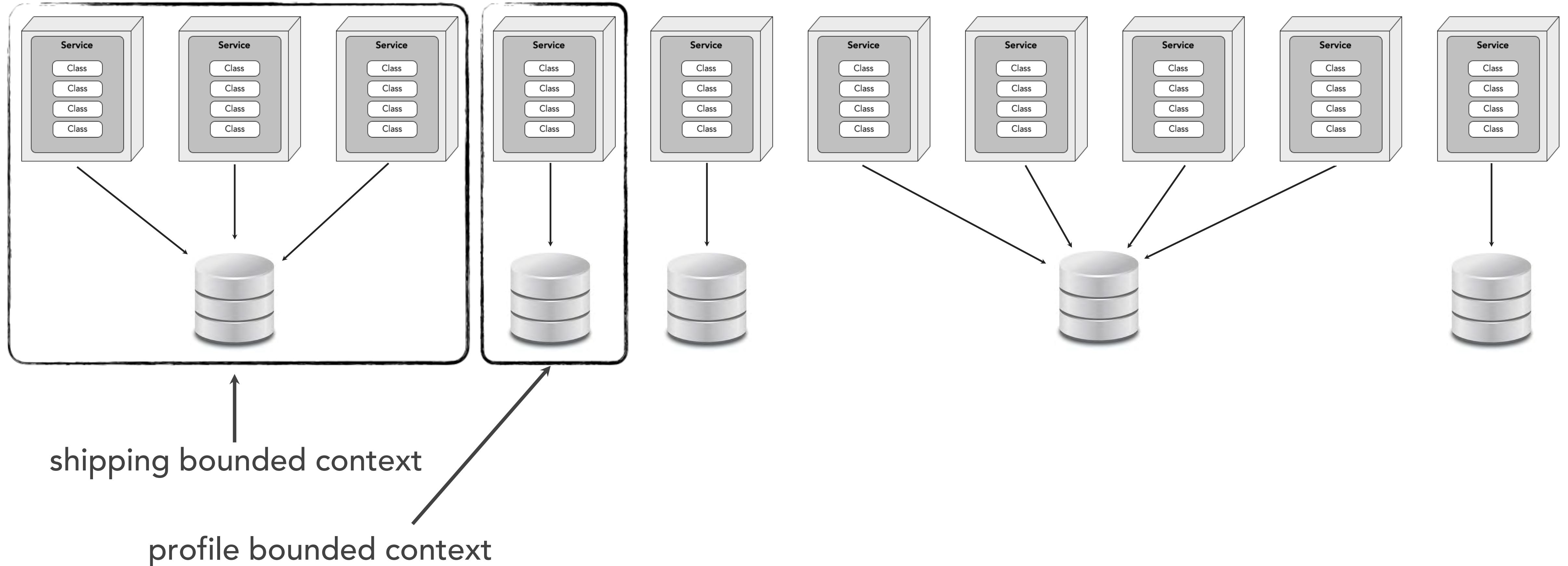
# bounded context



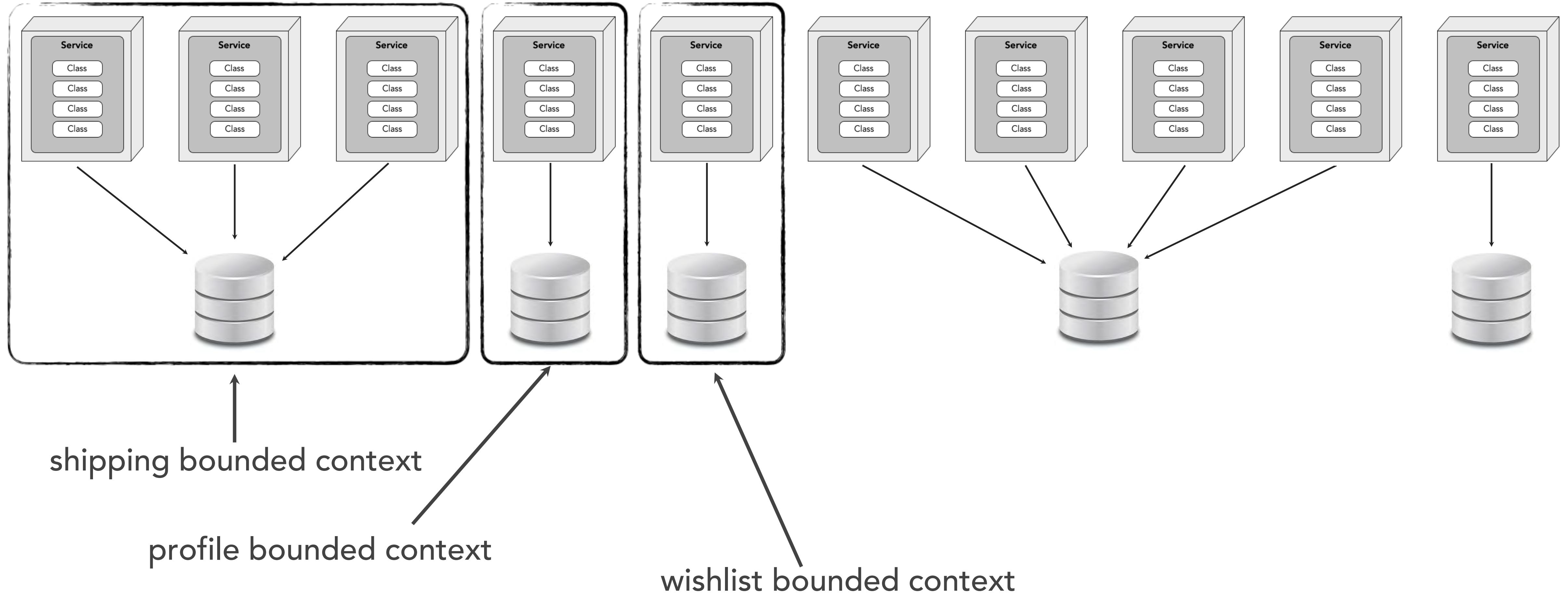
# bounded context



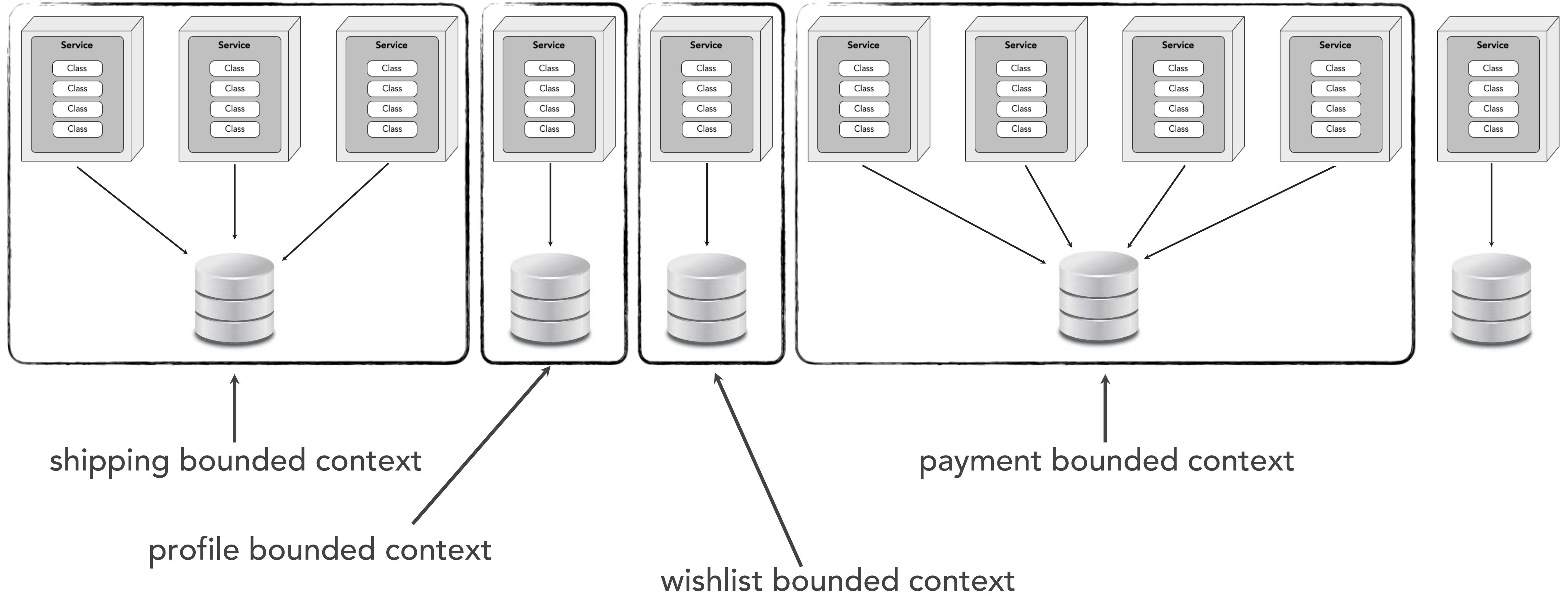
# bounded context



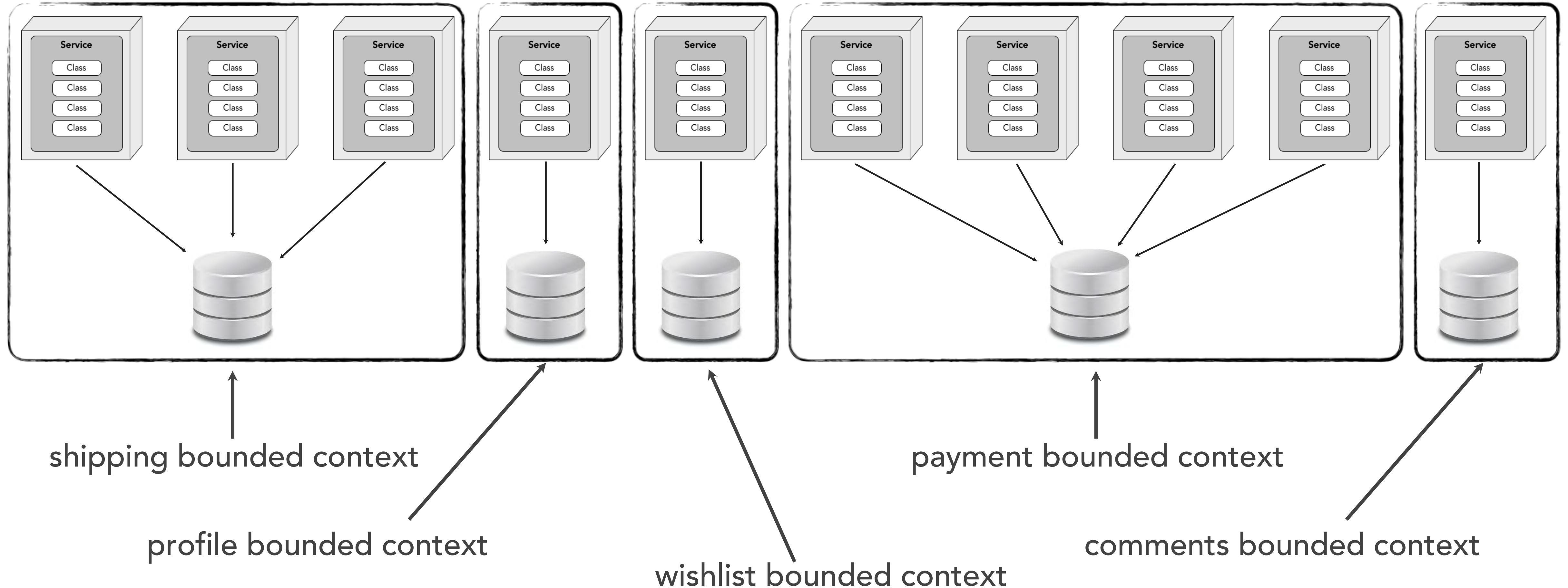
# bounded context



# bounded context



# bounded context



# bounded context

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface BoundedContext {
    public Context value();

    public enum Context {
        SHIPPING,
        PAYMENT,
        WISHLIST,
        COMMENTS,
        PROFILE
    }
}
```

# bounded context

```
@ServiceEntrypoint  
@FunctionalService  
@BoundedContext(Context.PAYMENT)  
public class CreditCardServiceAPI {  
    ...  
}
```

# bounded context

```
@ServiceEntrypoint  
@FunctionalService  
@BoundedContext(Context.PAYMENT)  
public class CreditCardServiceAPI {
```

...

```
}
```

```
@ServiceEntrypoint  
@FunctionalService  
@BoundedContext(Context.PAYMENT)  
public class GiftCardServiceAPI {
```

...

```
}
```

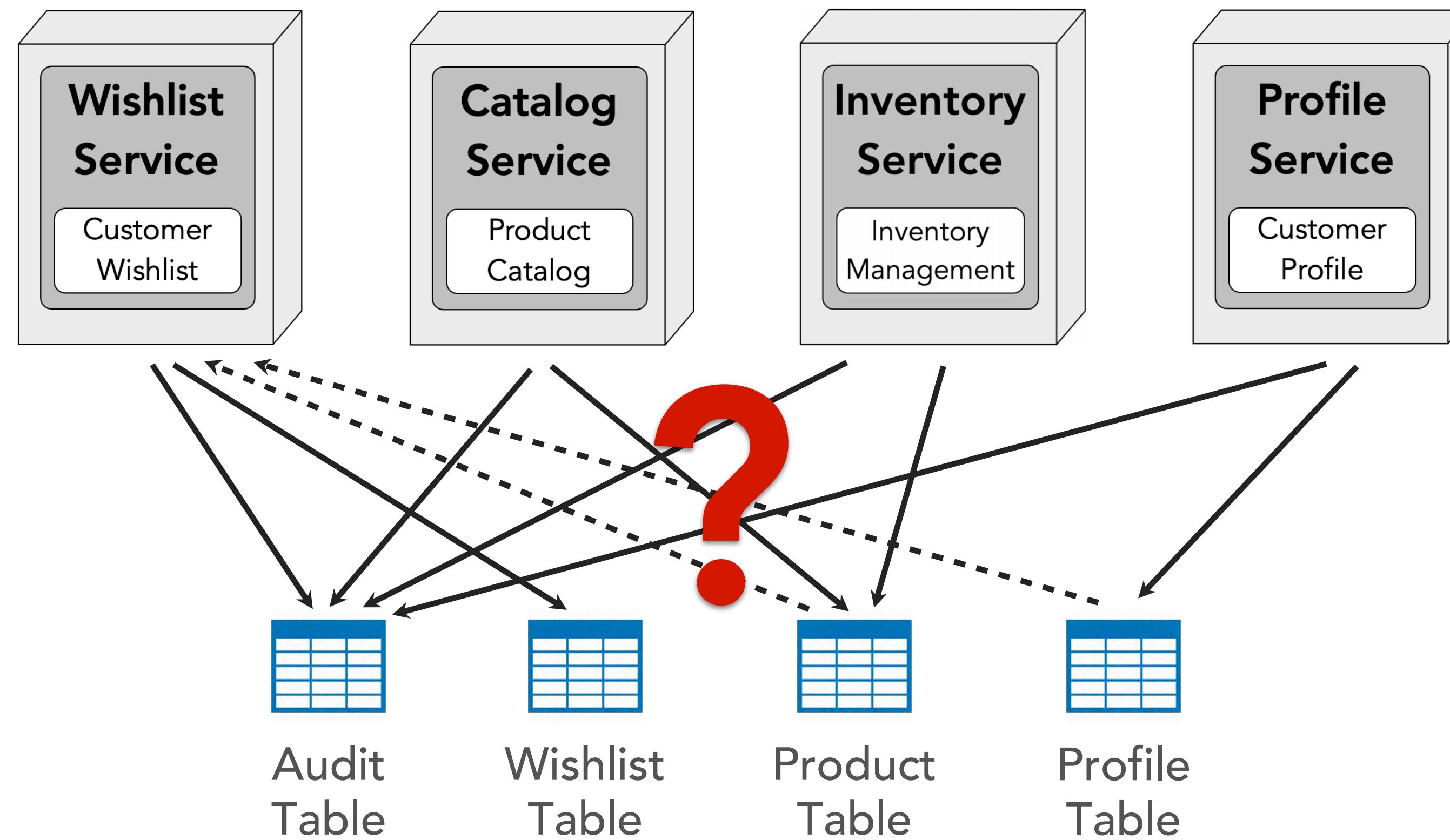
# bounded context

```
@ServiceEntrypoint  
@FunctionalService  
→ @BoundedContext(Context.PAYMENT)  
public class CreditCardServiceAPI {  
    ...  
}  
  
@ServiceEntrypoint  
@FunctionalService  
→ @BoundedContext(Context.PAYMENT)  
public class GiftCardServiceAPI {  
    ...  
}
```

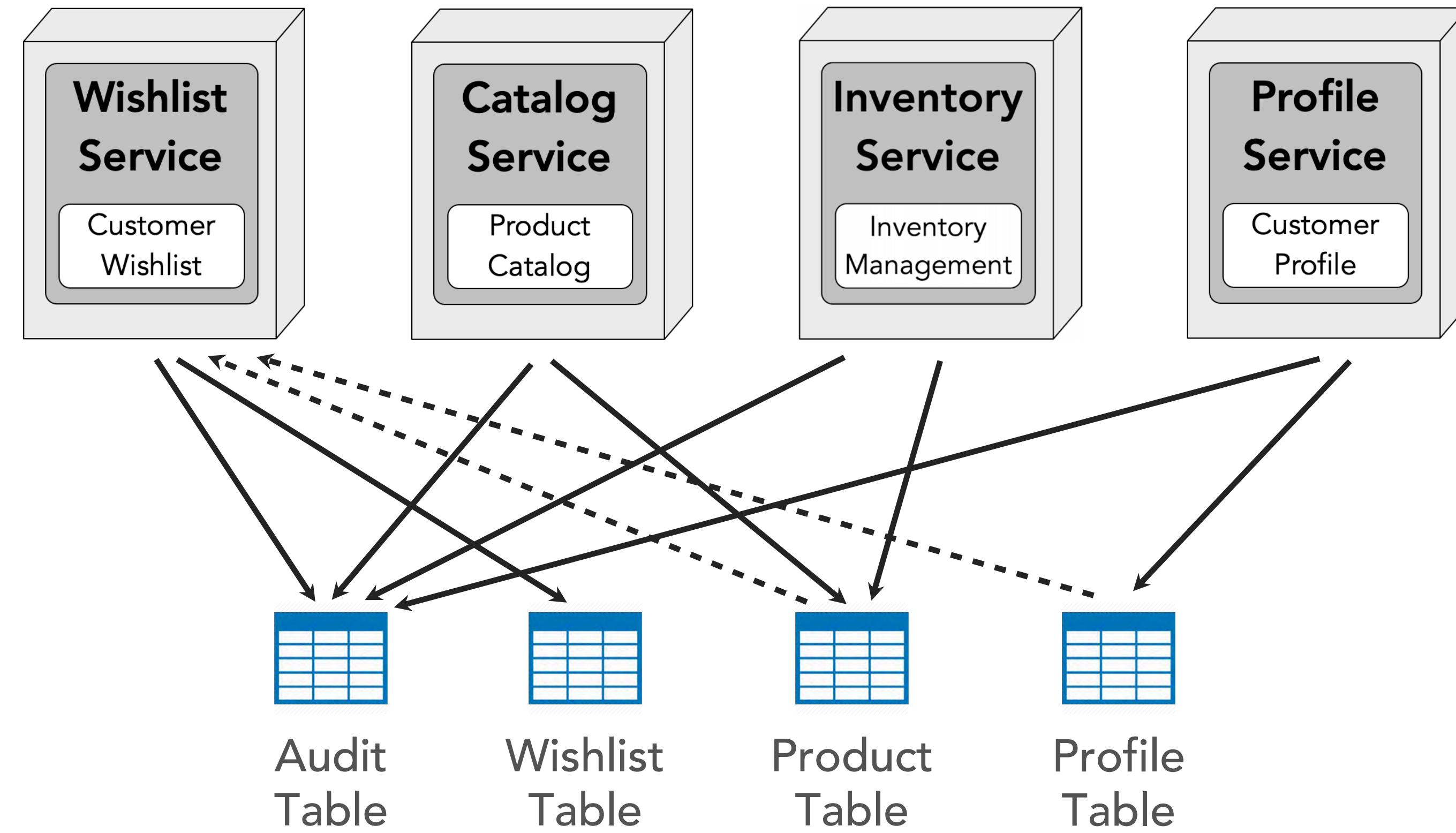
# Data Ownership and Access

# data ownership

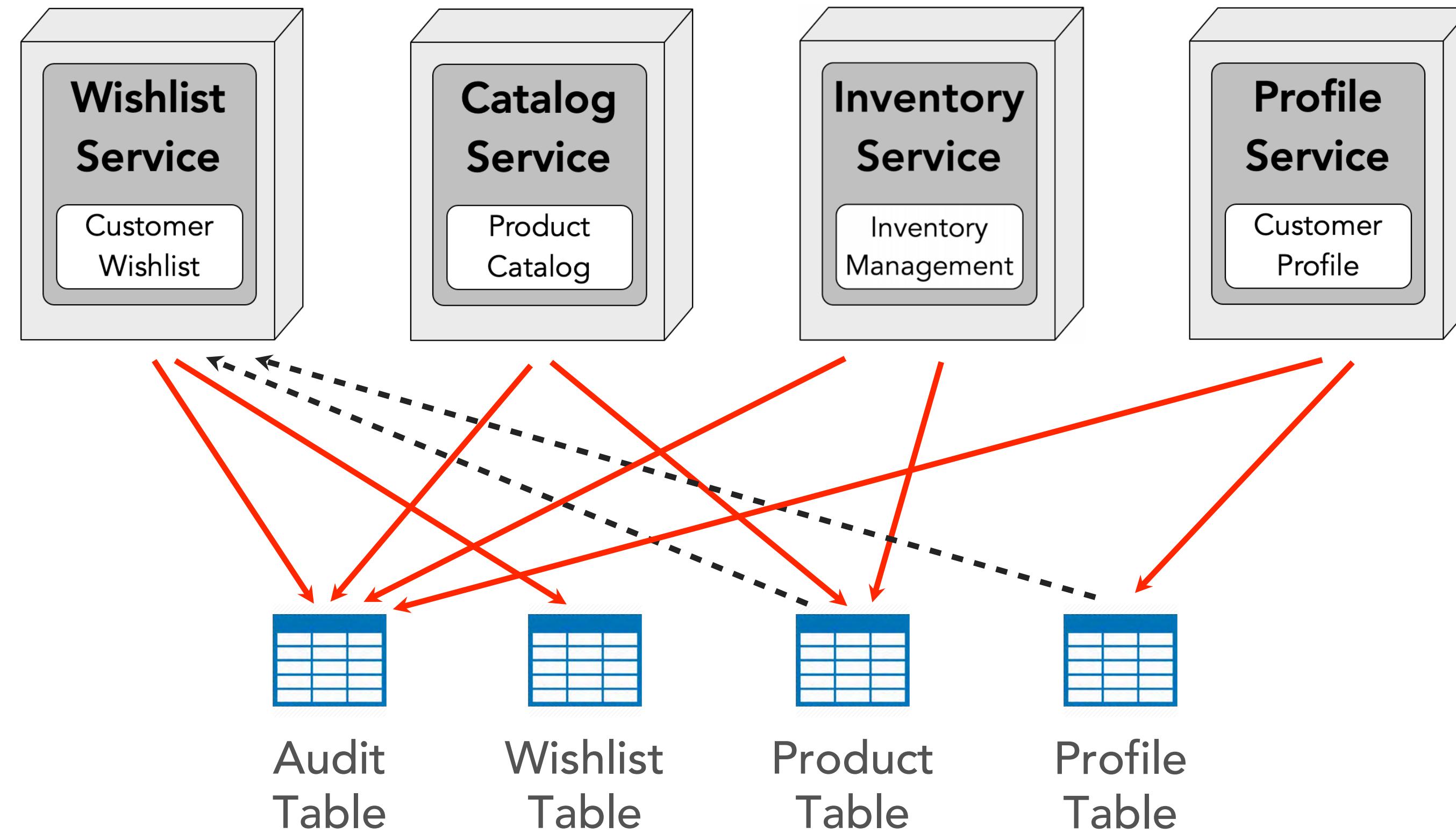
*“How do I assign table ownership to my services?”*



# data ownership

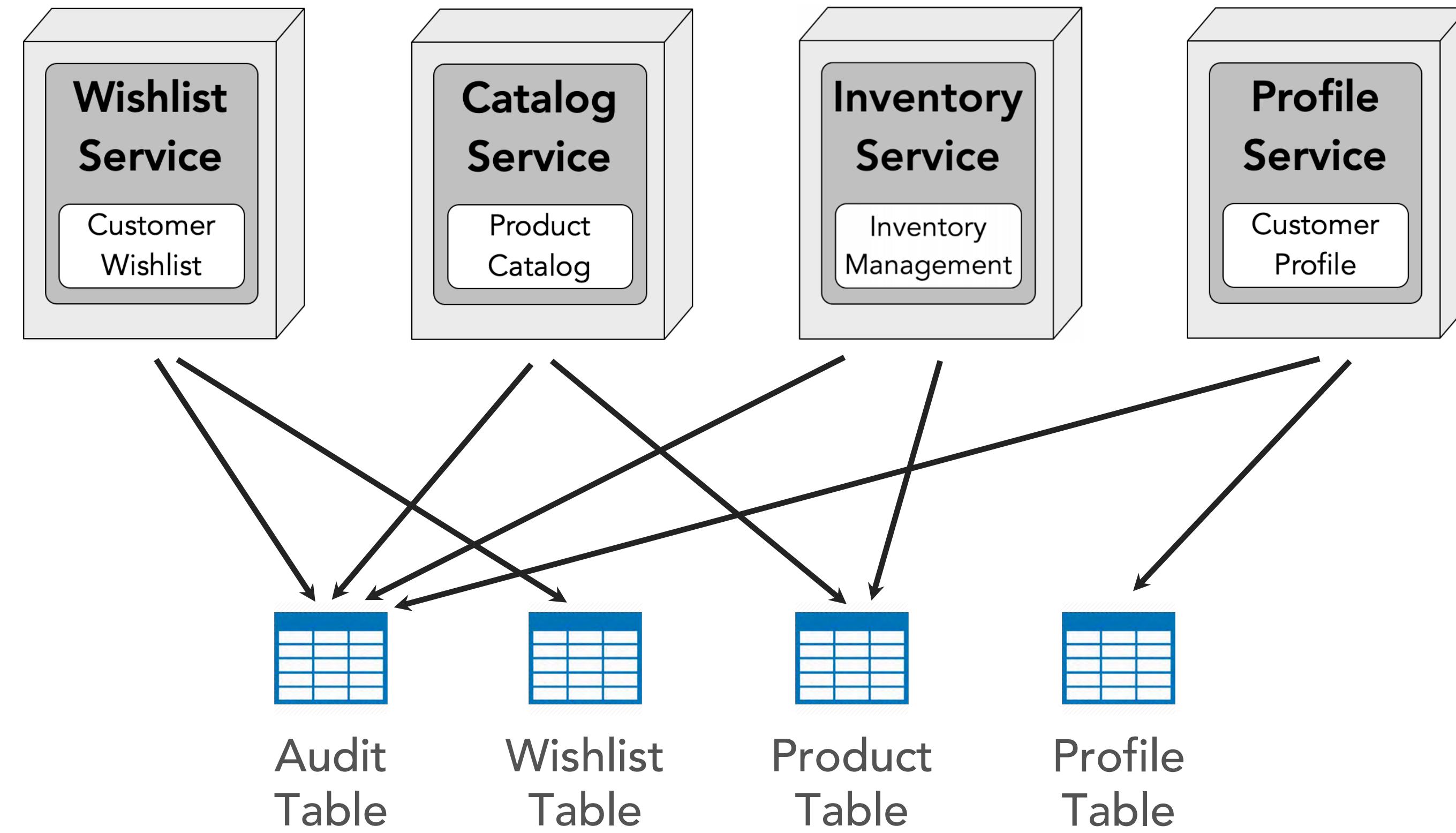


# data ownership

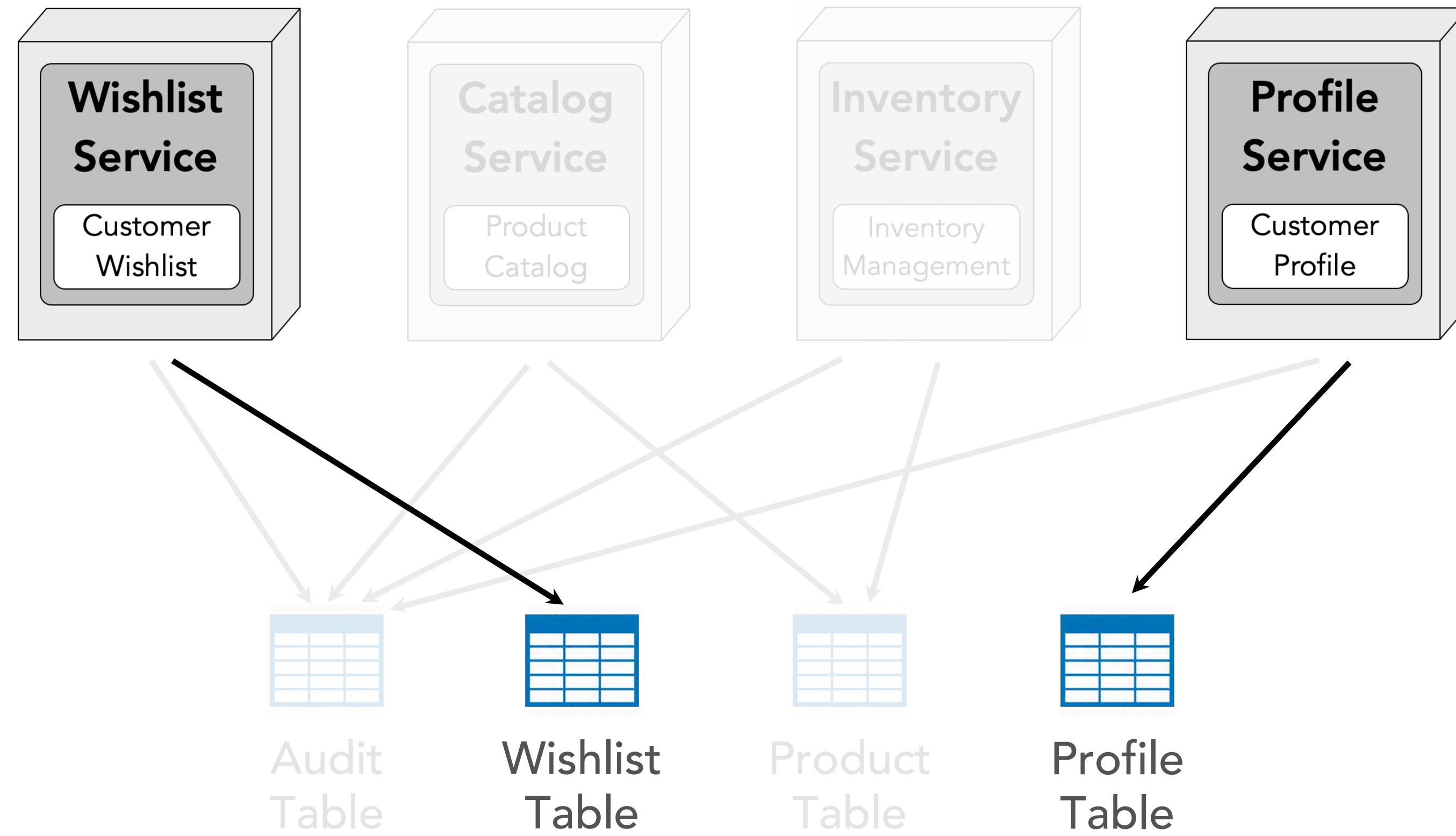


the owner is the one who writes to the table

# data ownership

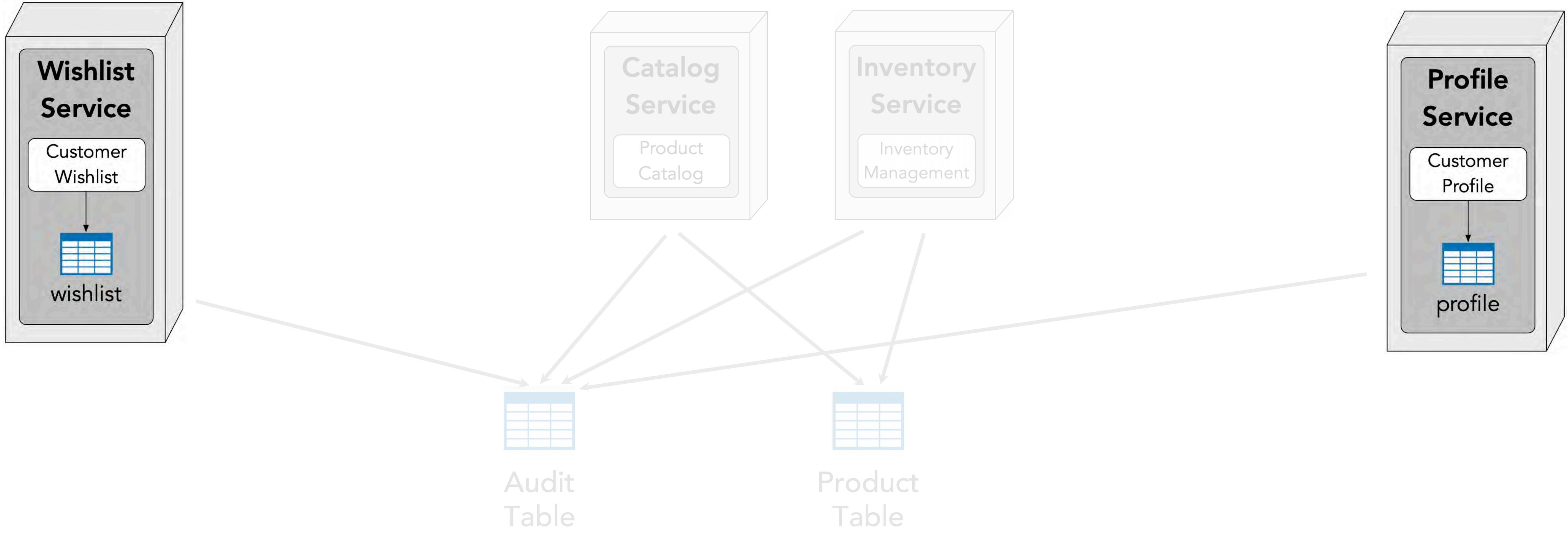


# data ownership



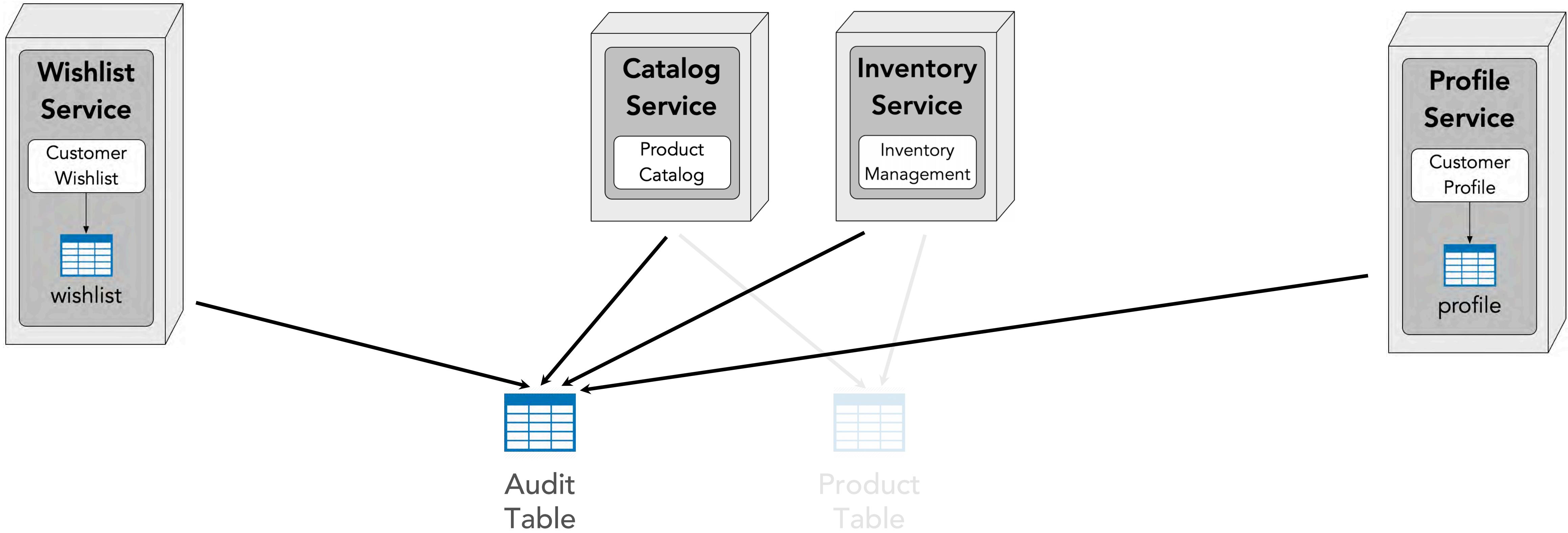
scenario: single ownership

# data ownership



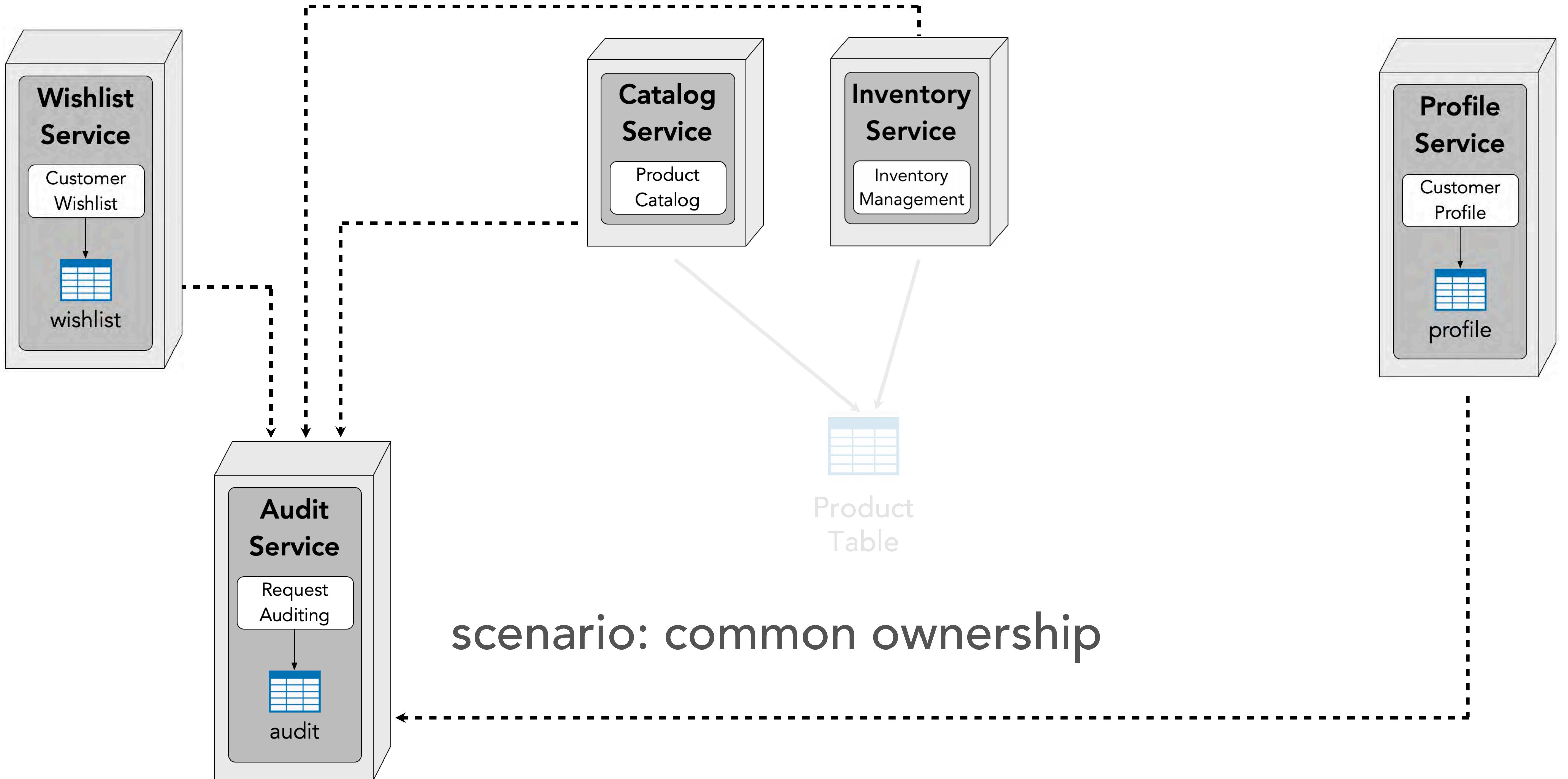
scenario: single ownership

# data ownership

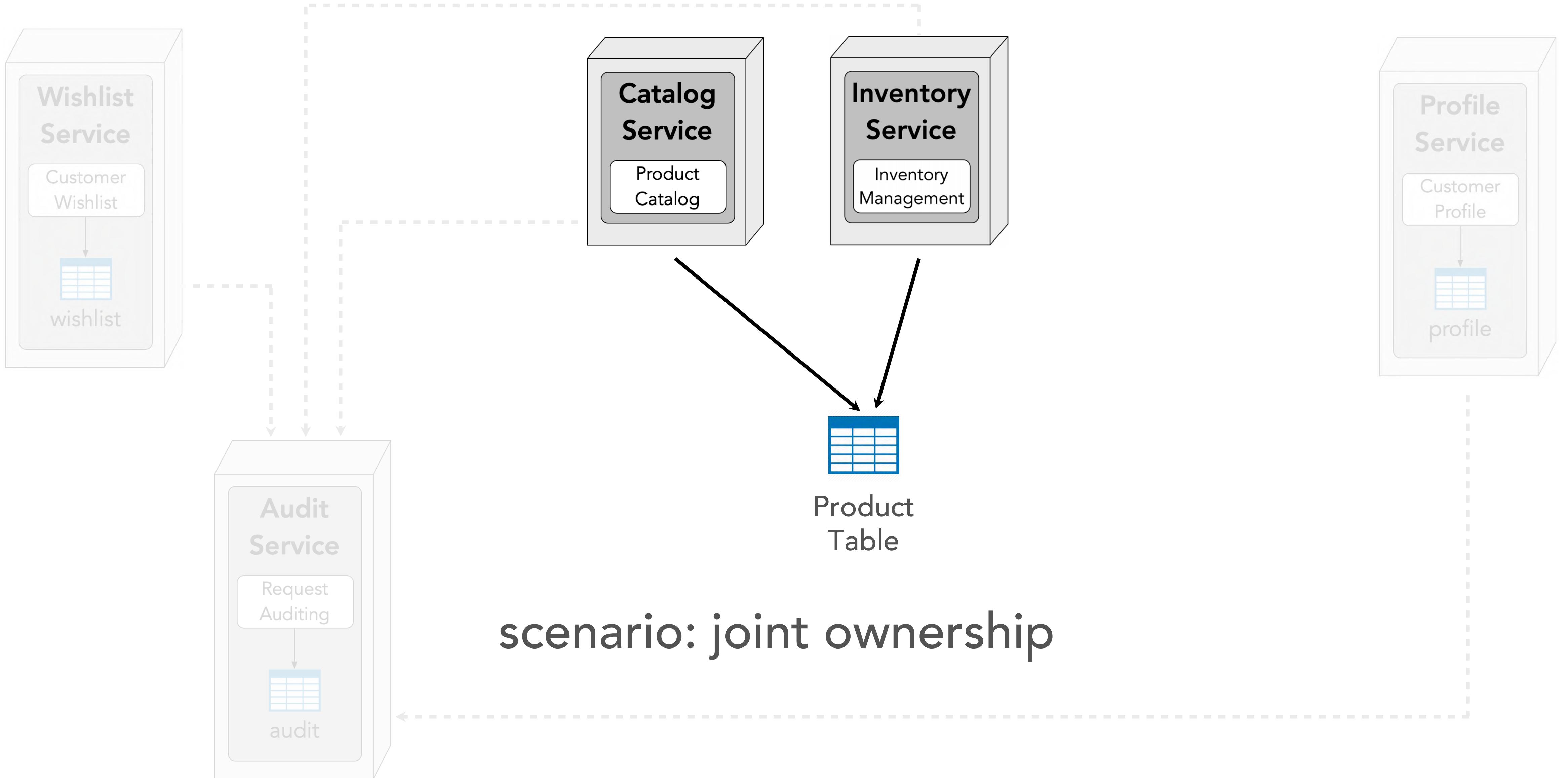


scenario: common ownership

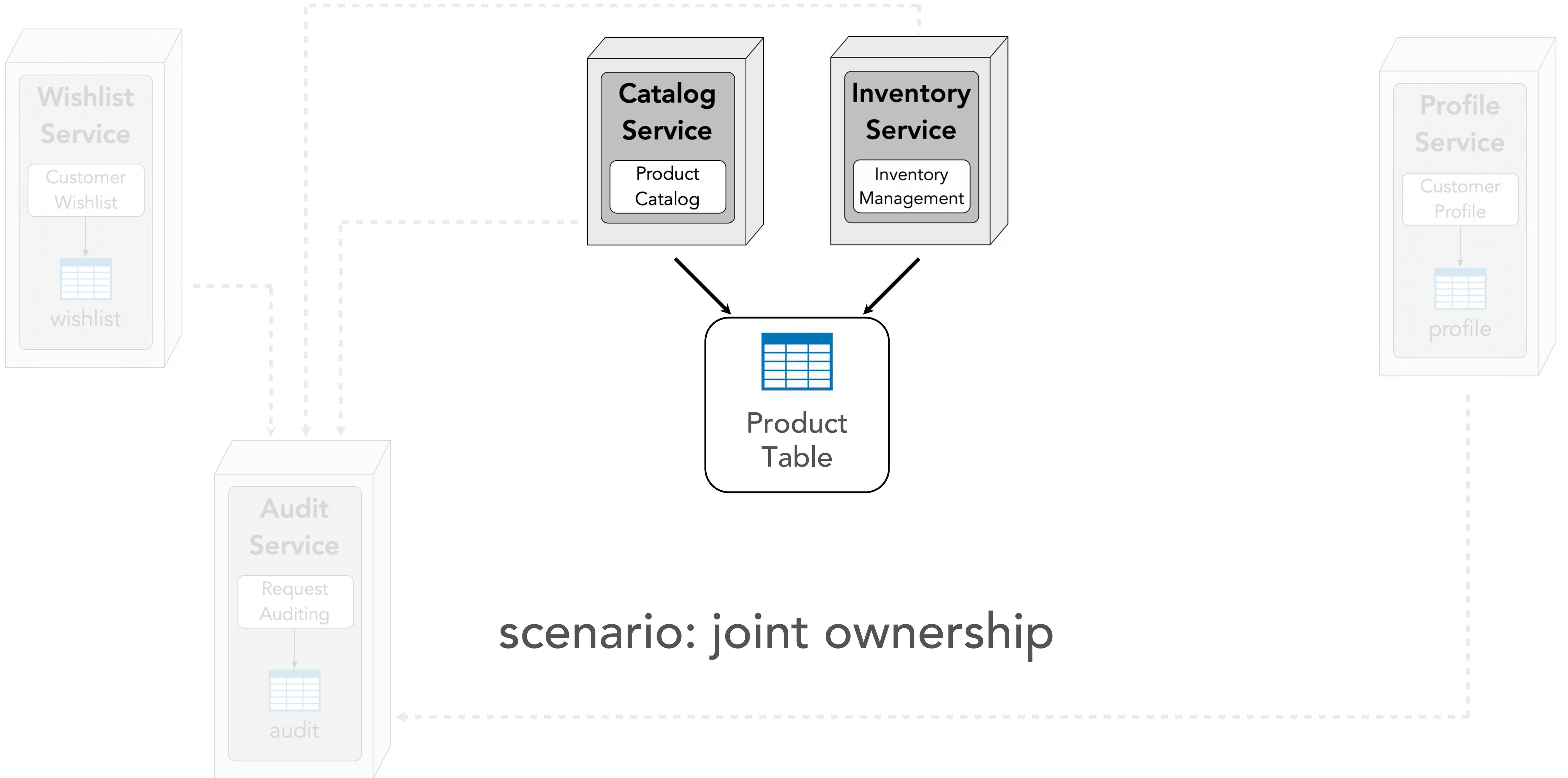
# data ownership



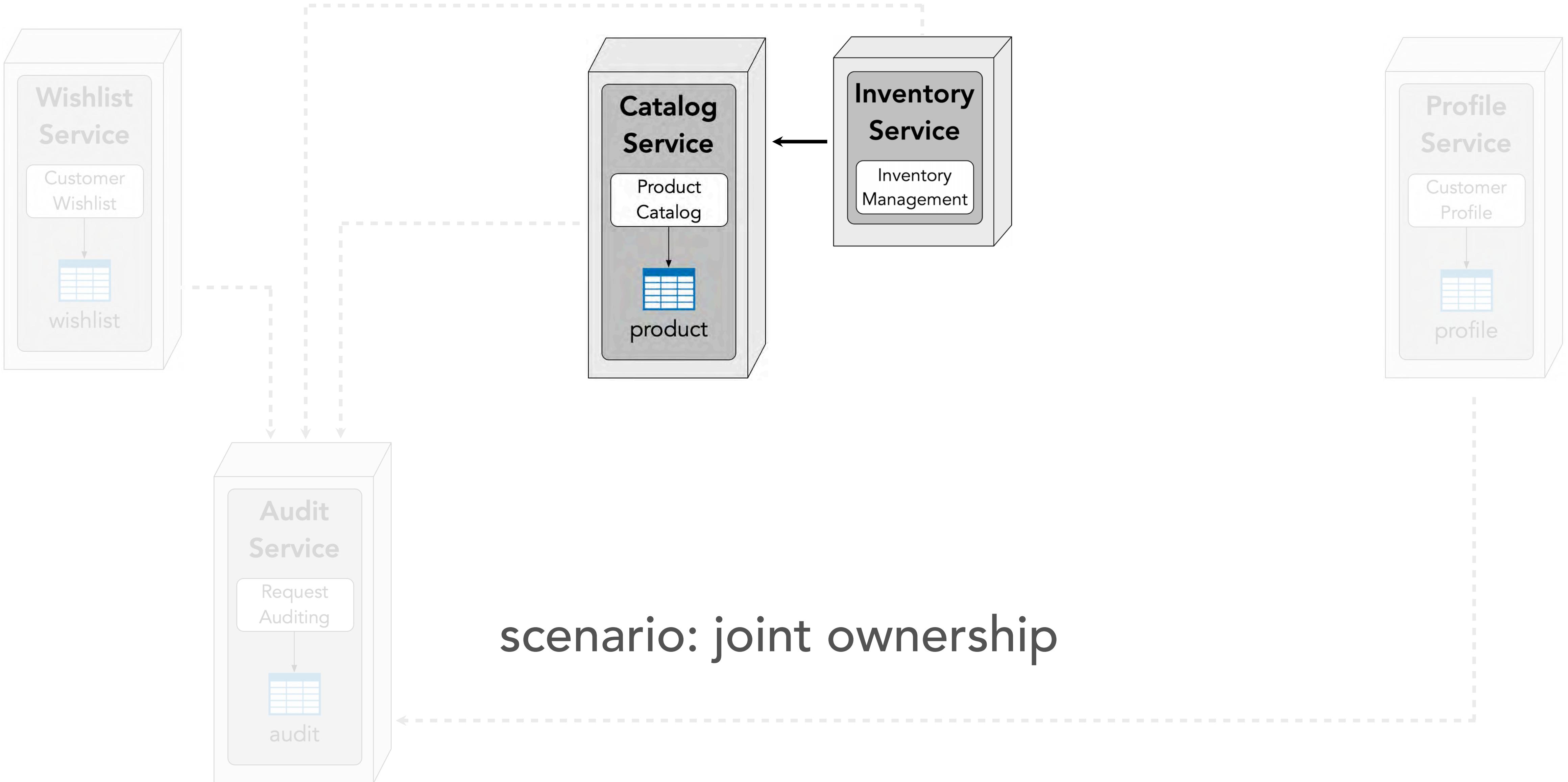
# data ownership



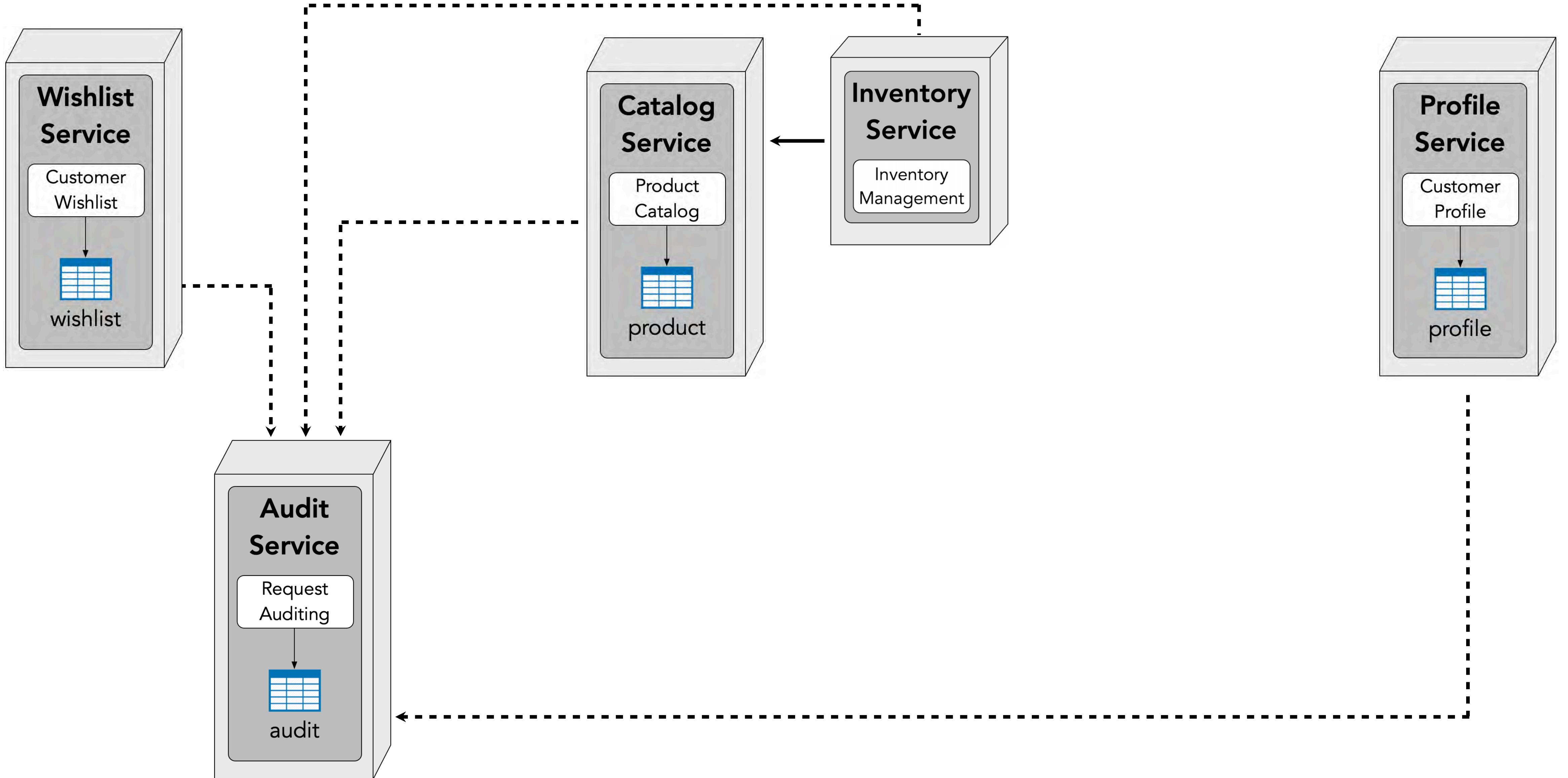
# data ownership



# data ownership

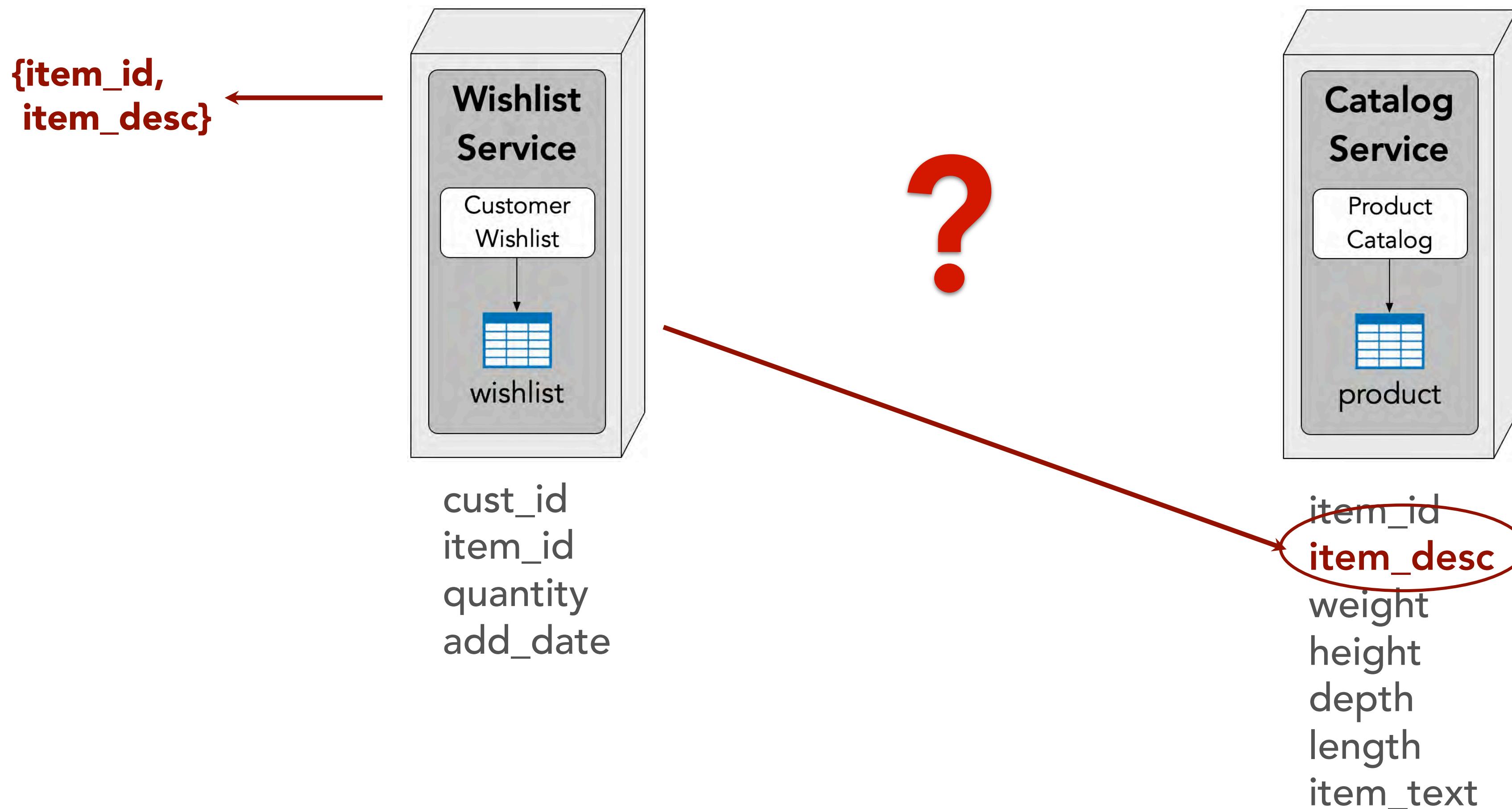


# data ownership



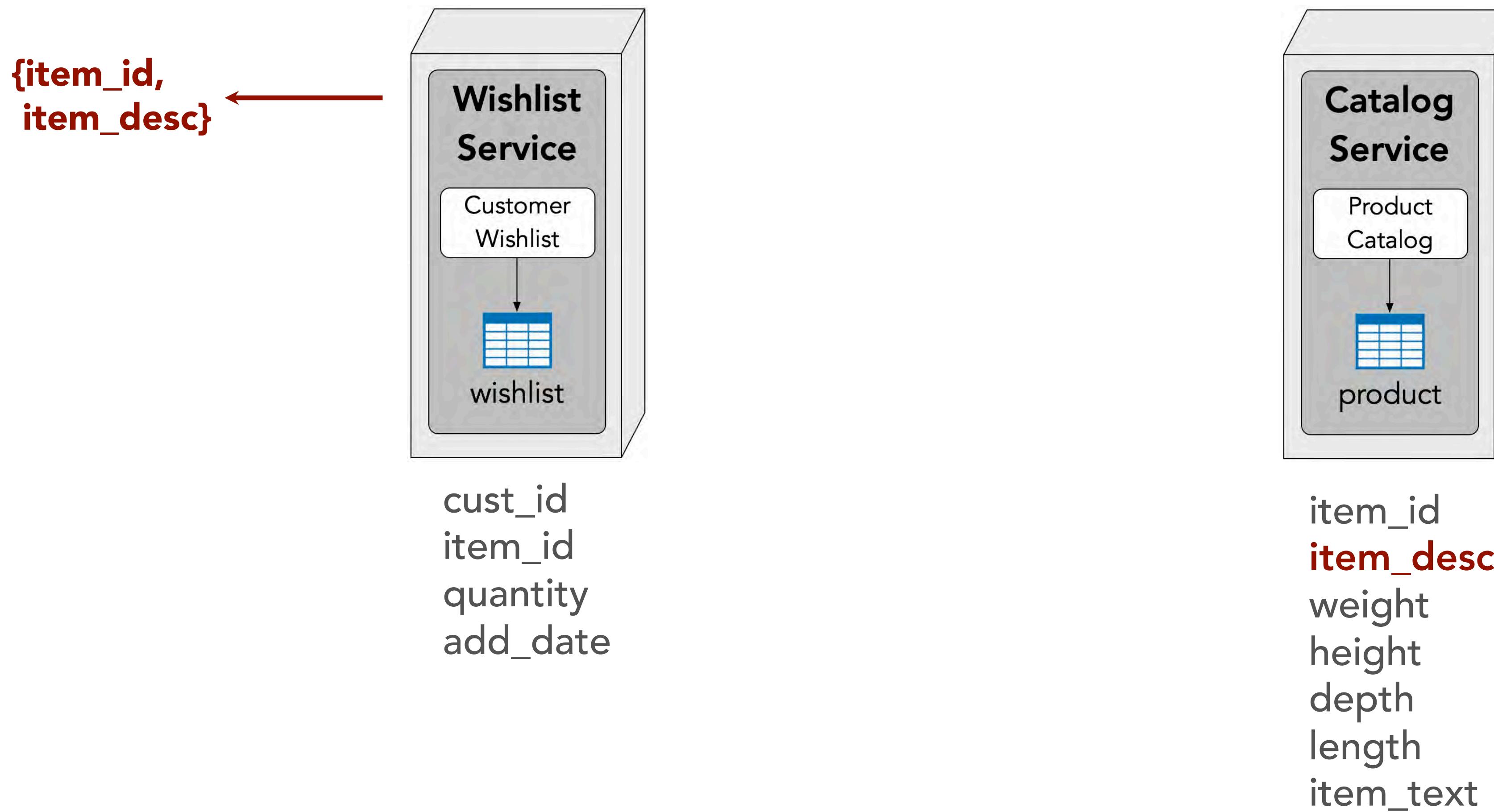
# data access

*“once I break apart my data, how do I access data I don’t own?”*



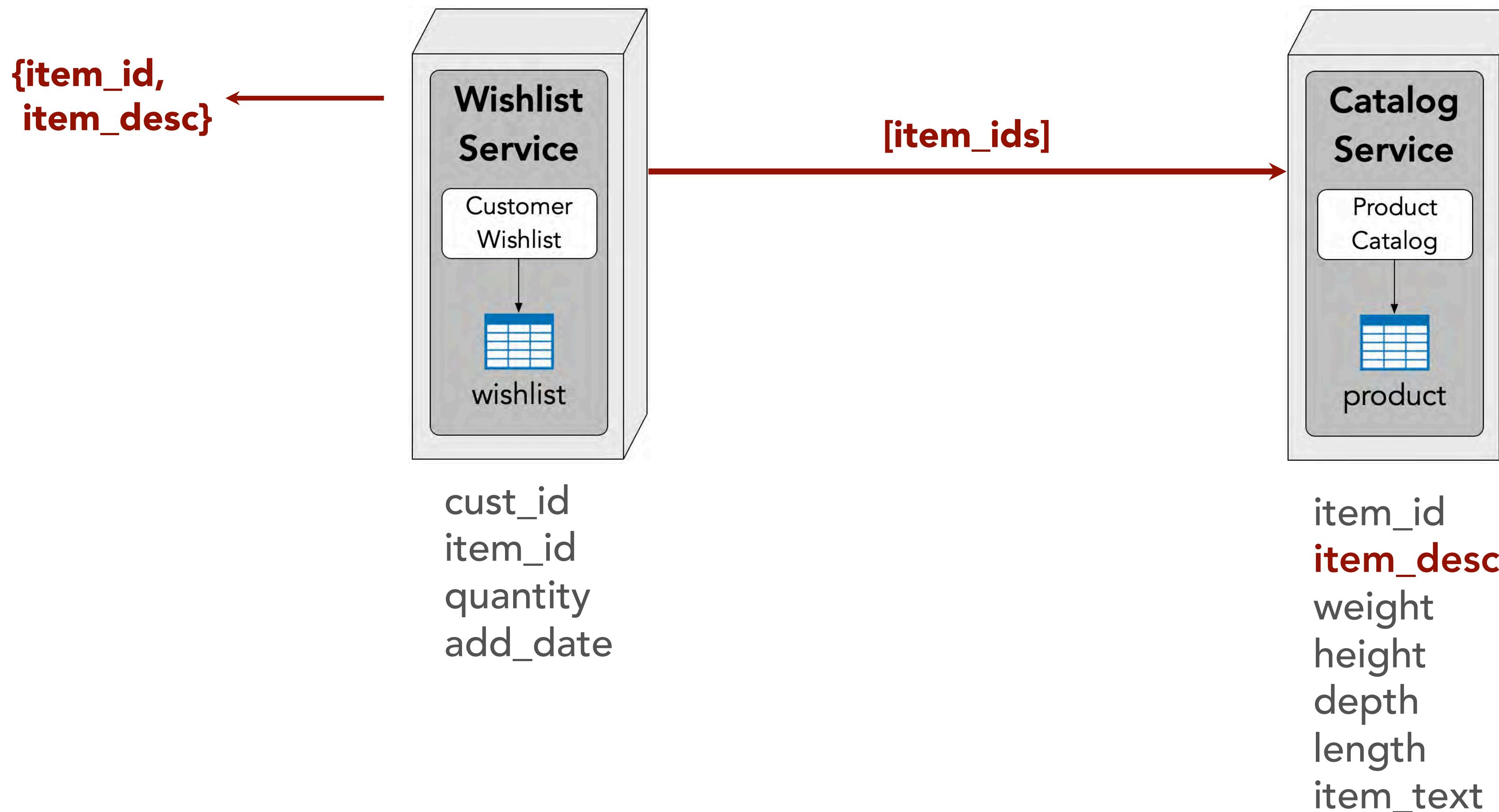
# data access

## option 1: interservice communication



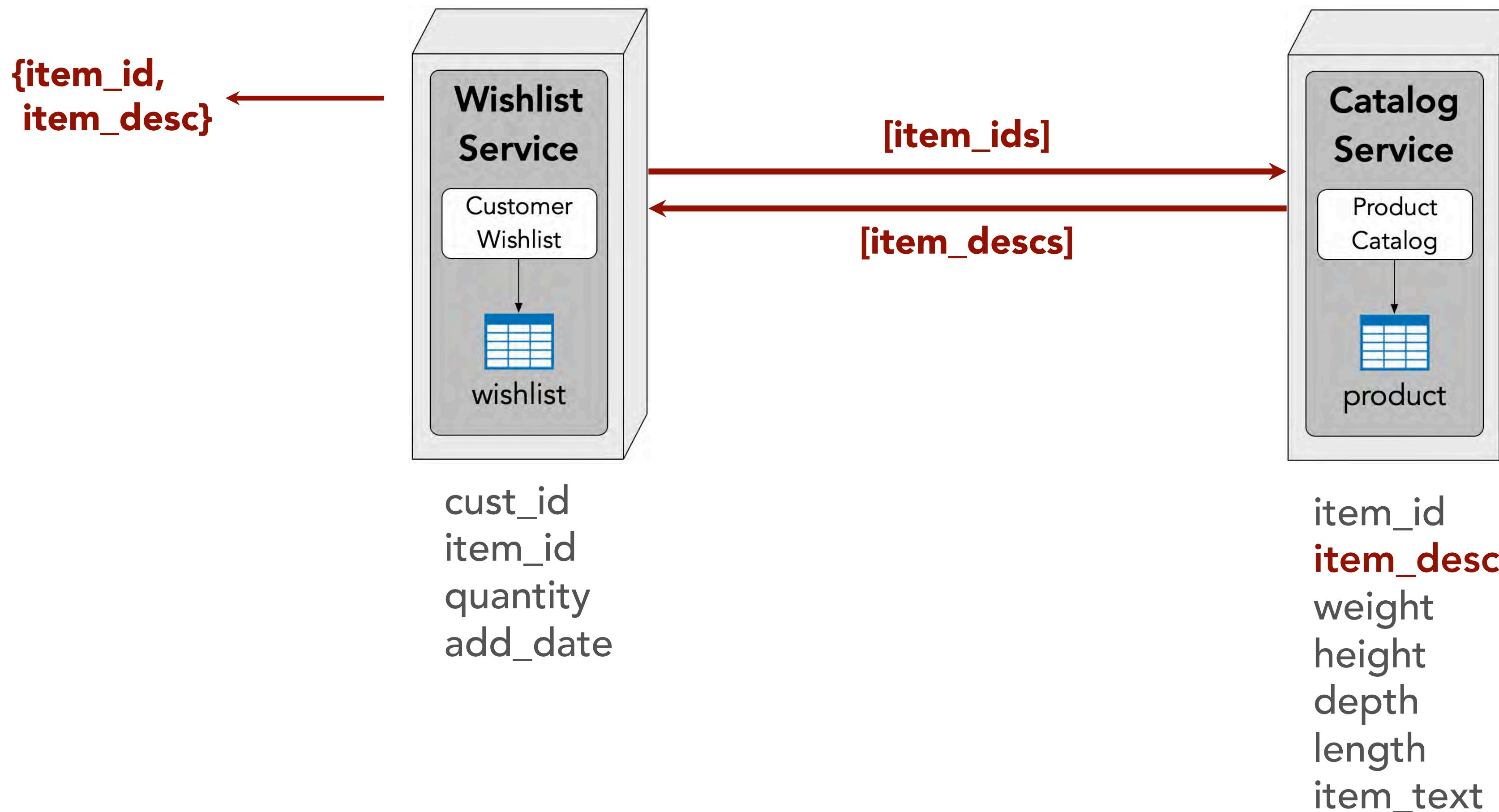
# data access

## option 1: interservice communication



# data access

## option 1: interservice communication

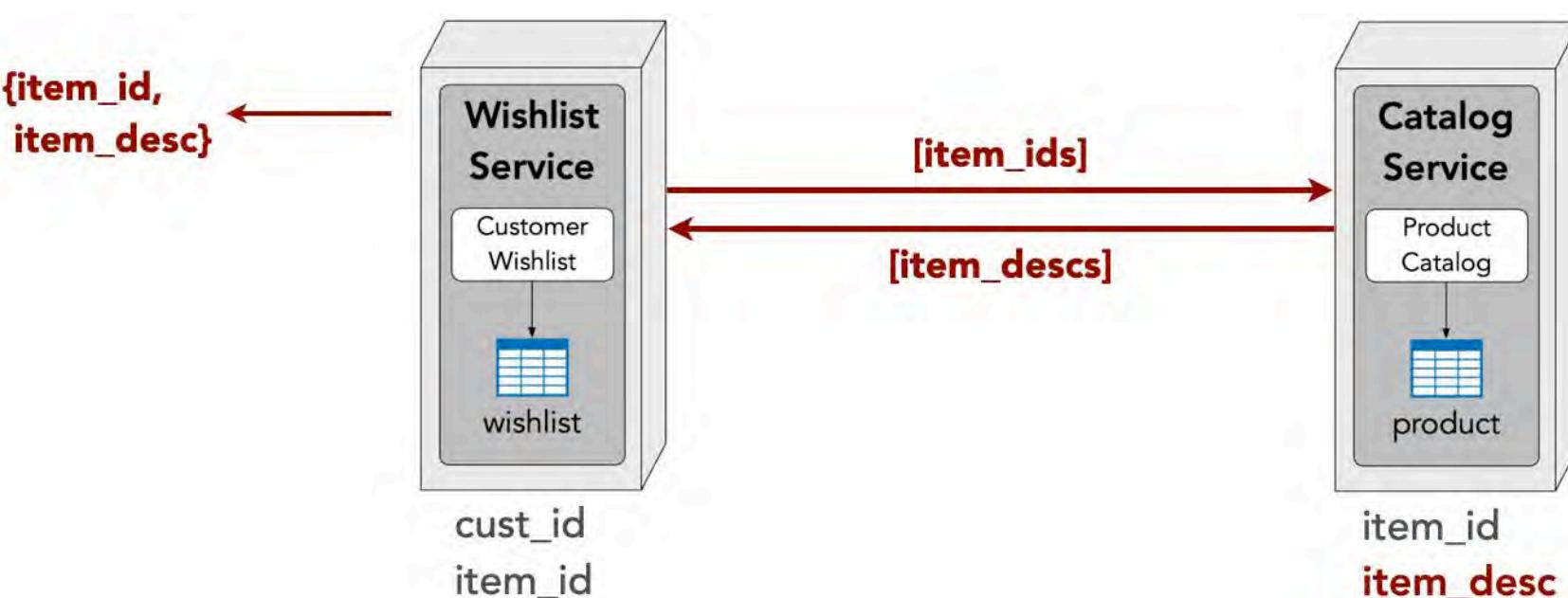


# tradeoffs

## option 1: interservice communication

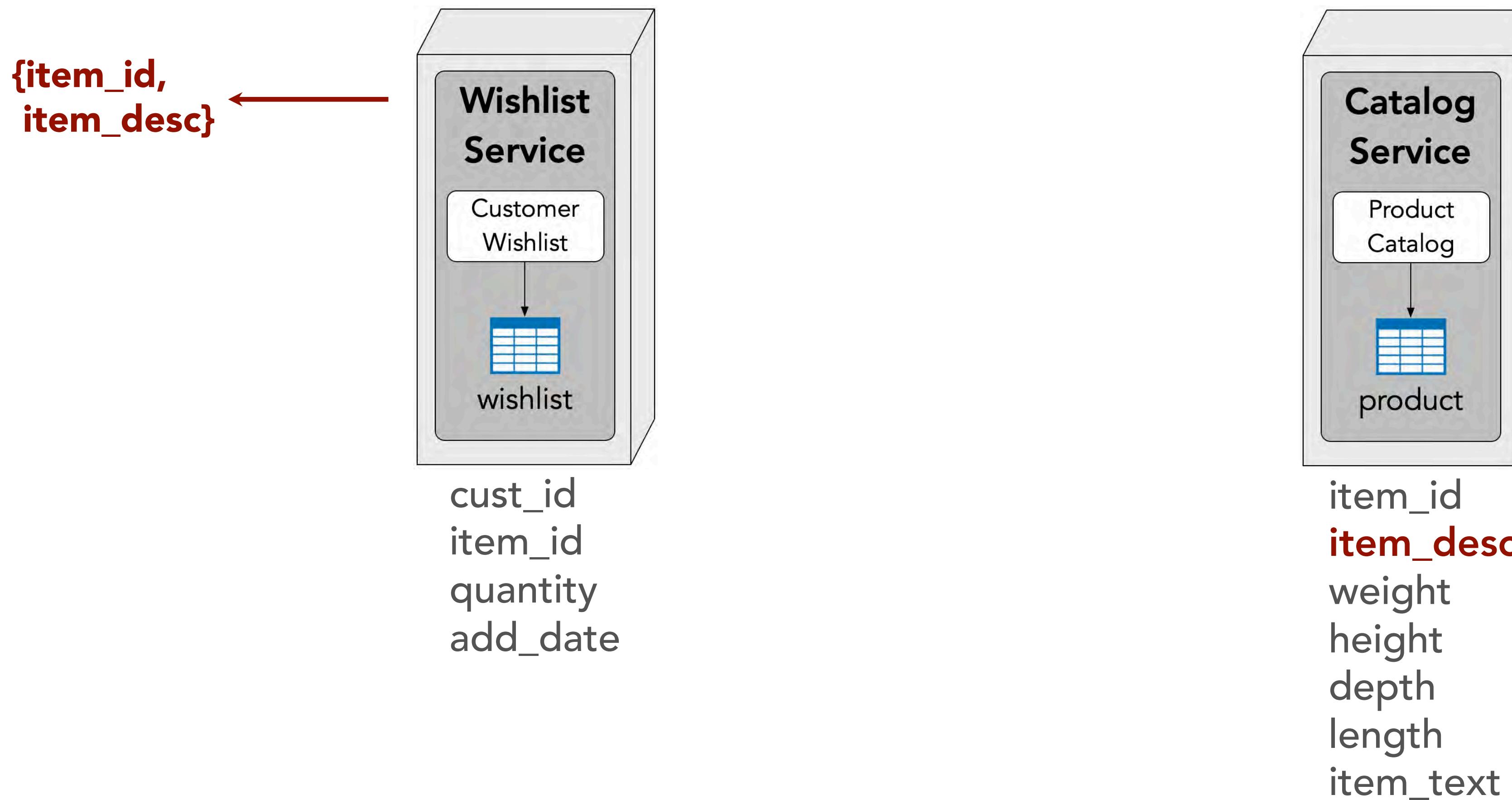


- ! network and security latency
- ! scalability and throughput
- ! fault tolerance



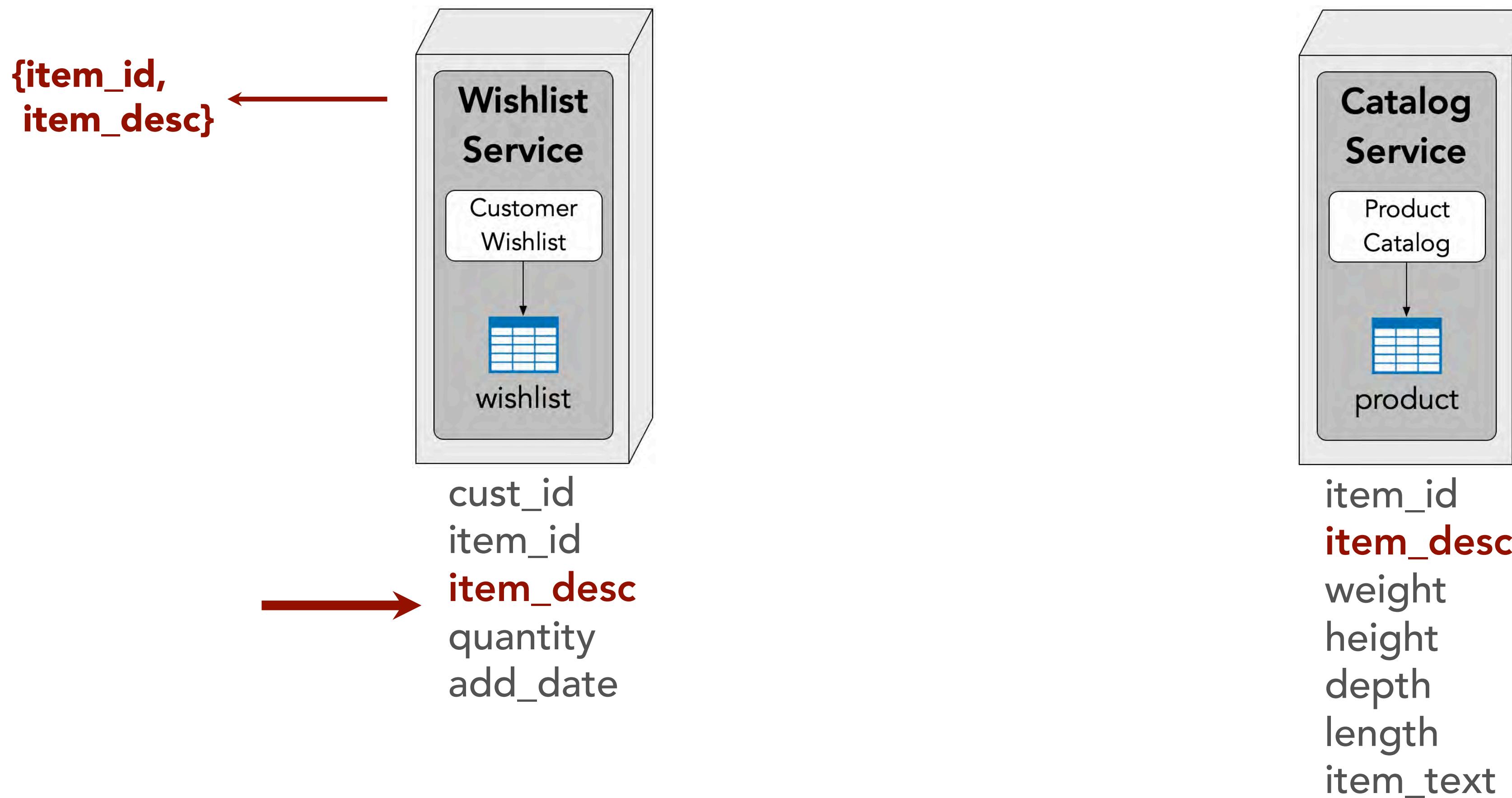
# data access

## option 2: data replication



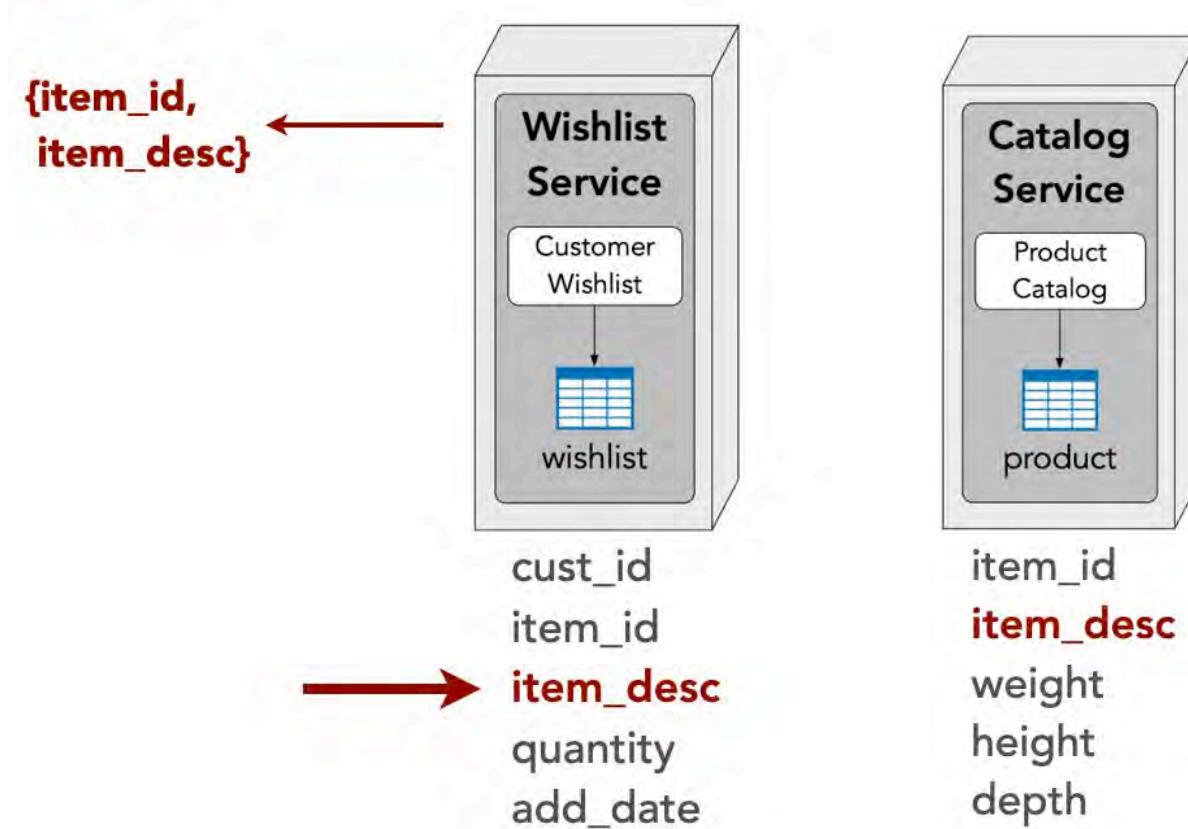
# data access

## option 2: data replication



# tradeoffs

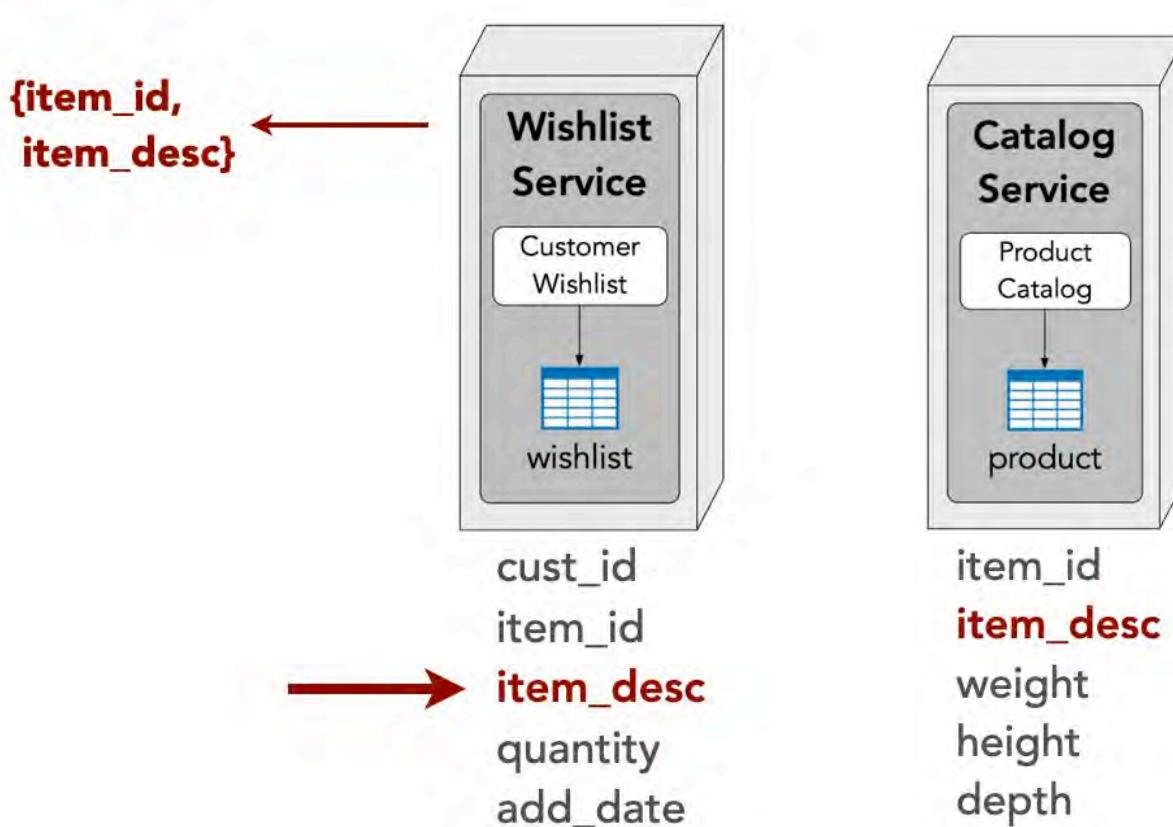
## option 2: data replication



# tradeoffs

## option 2: data replication

- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance



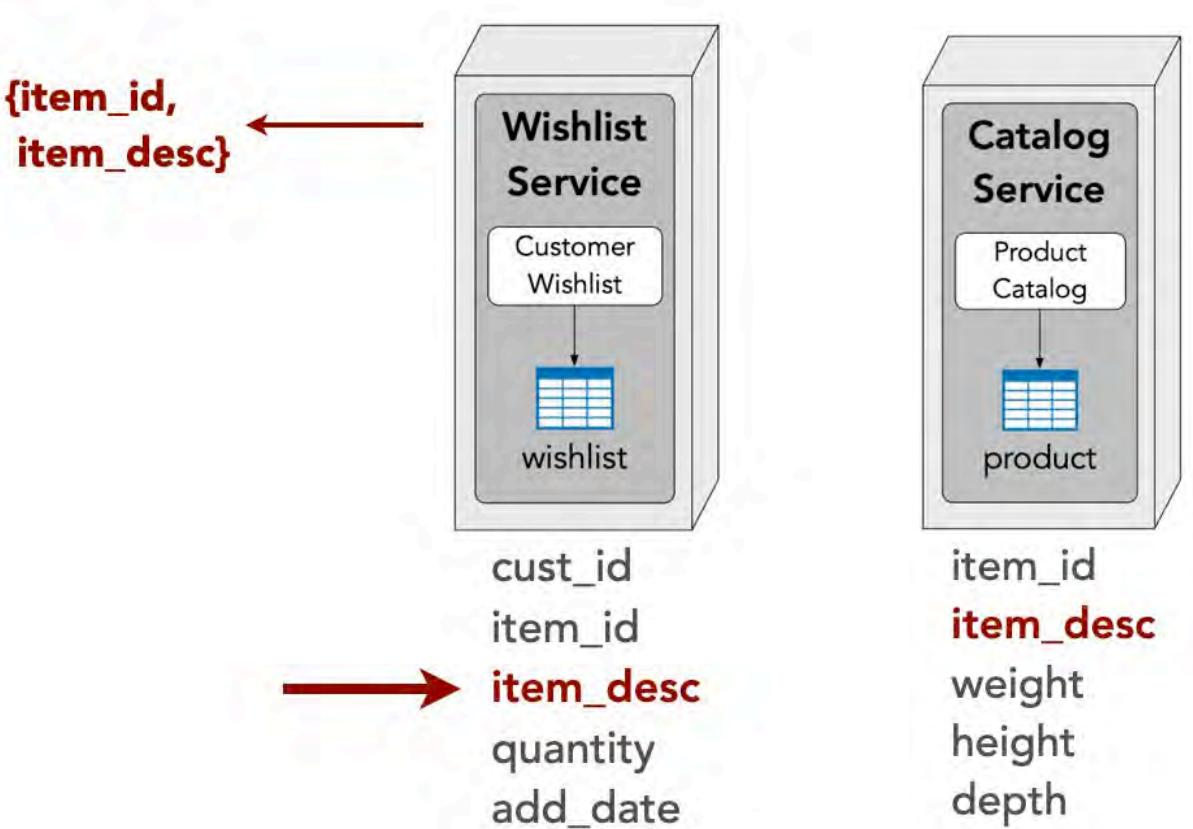
# tradeoffs

## option 2: data replication

- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance

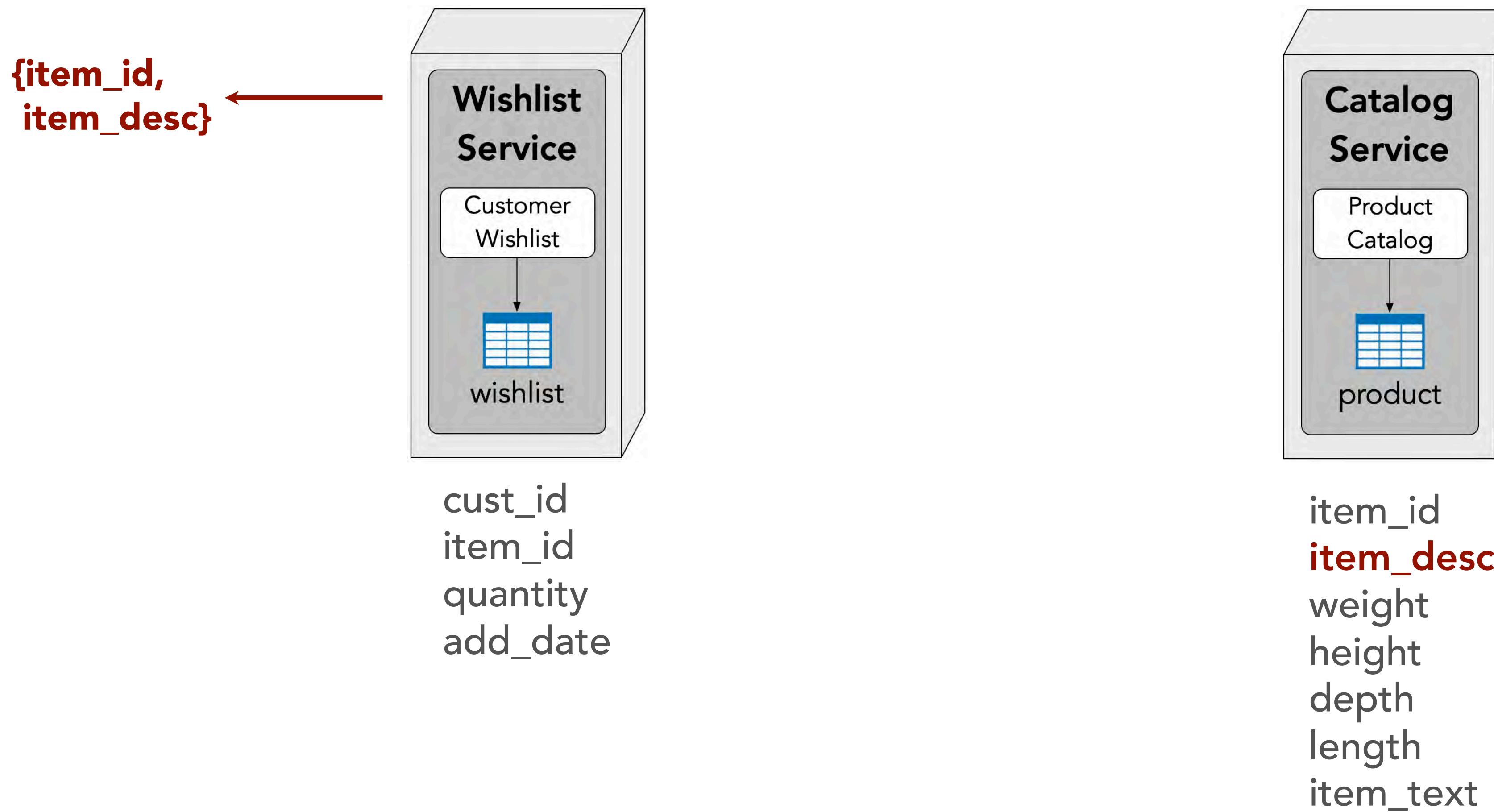


- ! data consistency issues
- ! data ownership issues



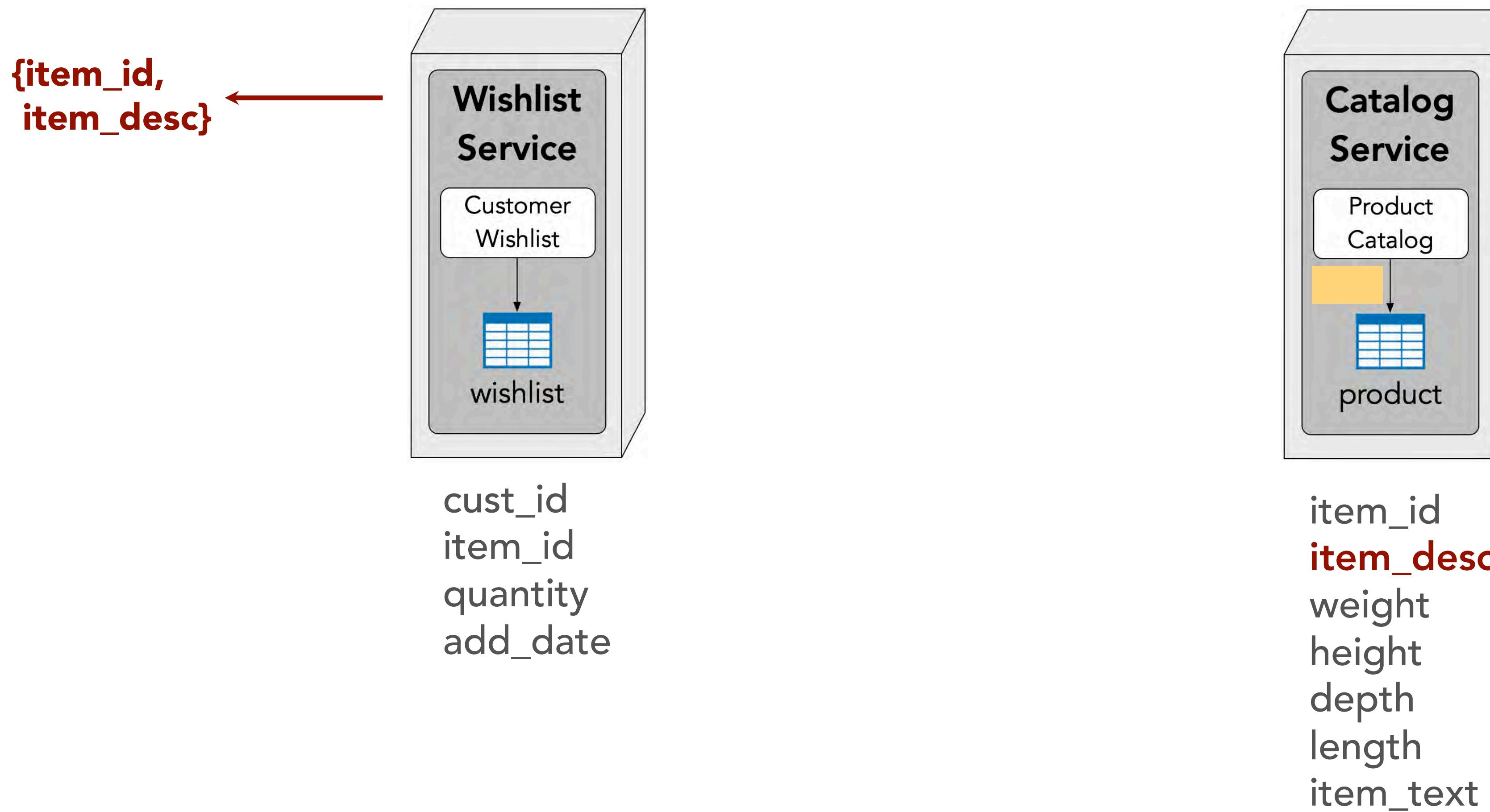
# data access

option 3: in-memory replicated cache



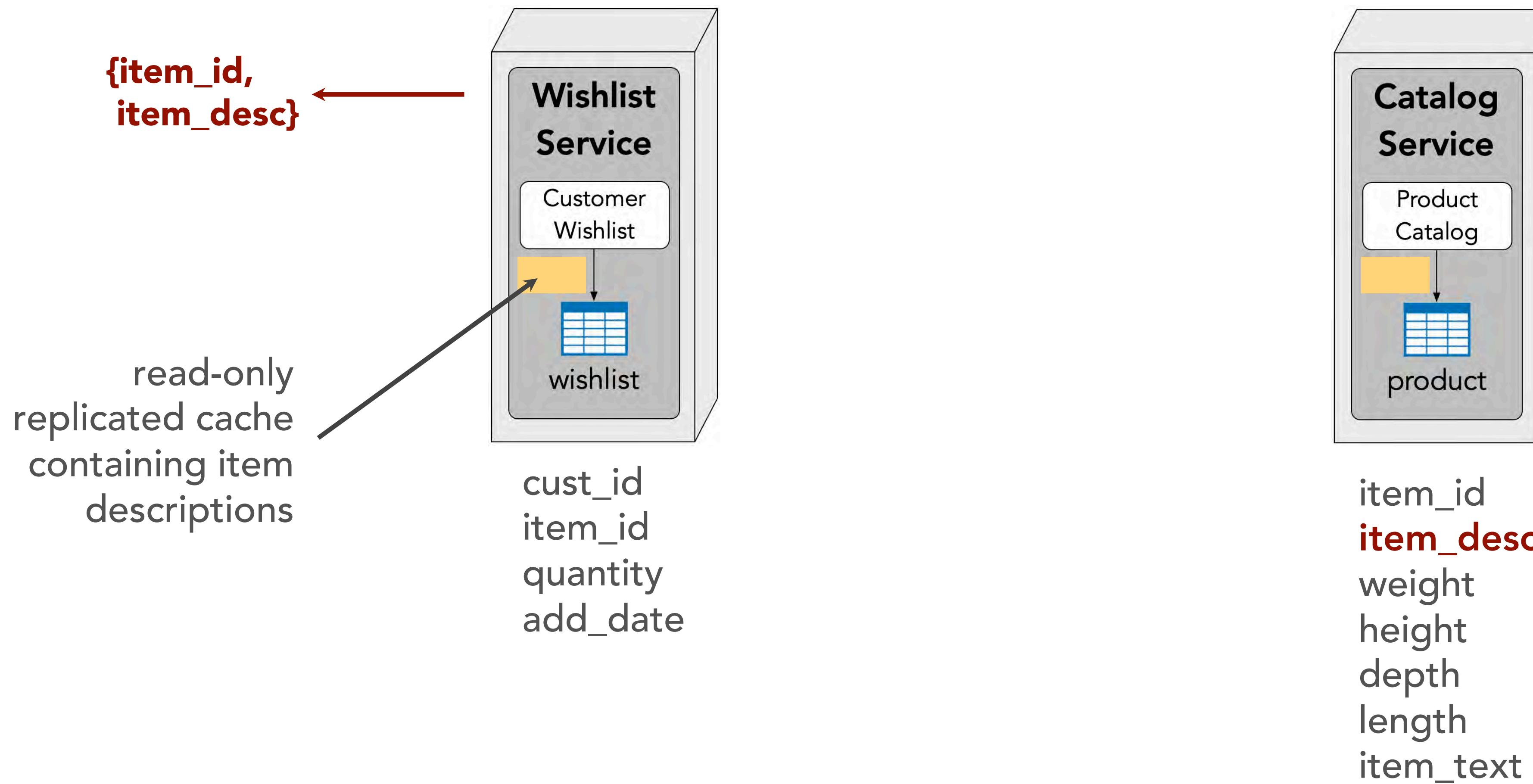
# data access

option 3: in-memory replicated cache



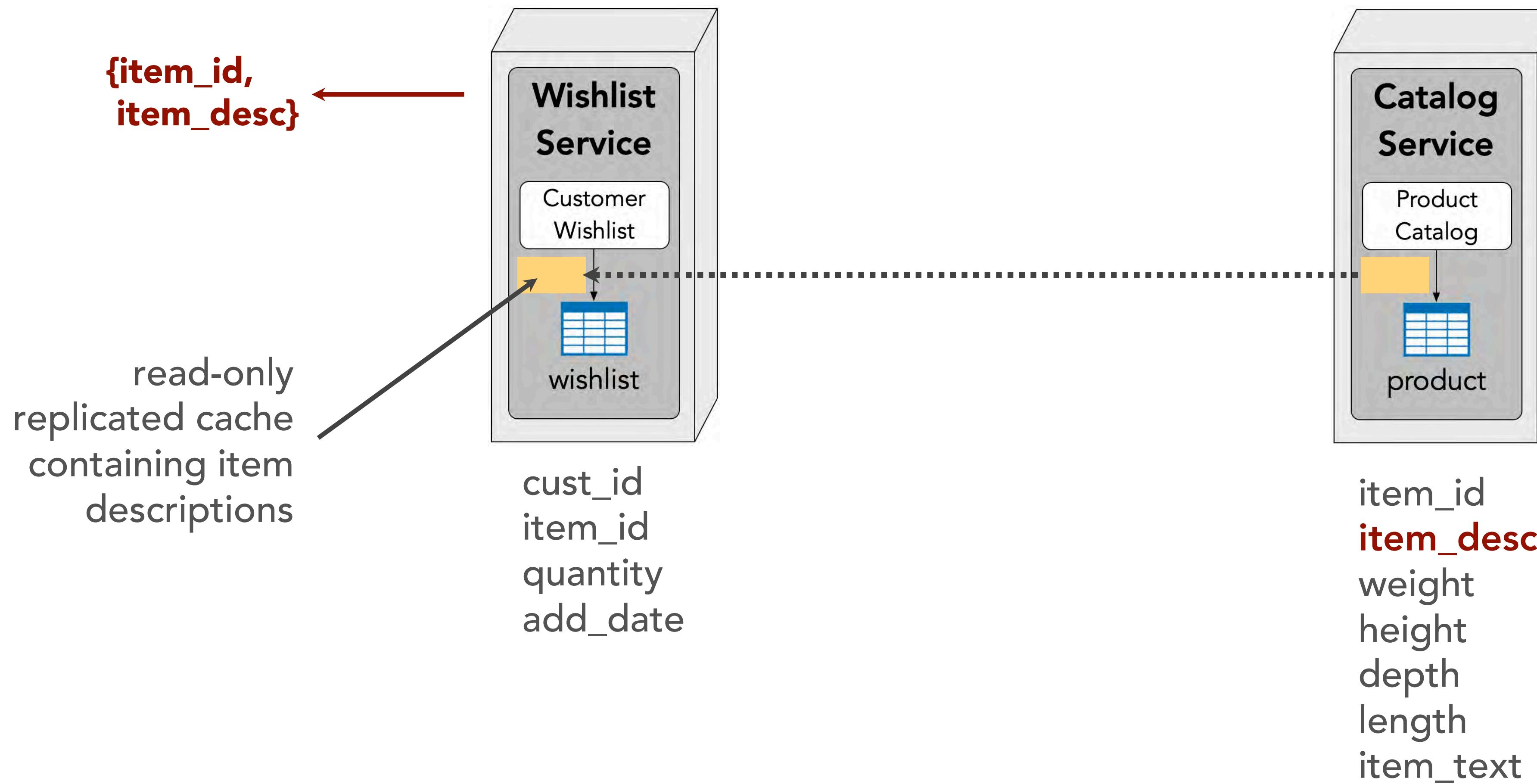
# data access

## option 3: in-memory replicated cache



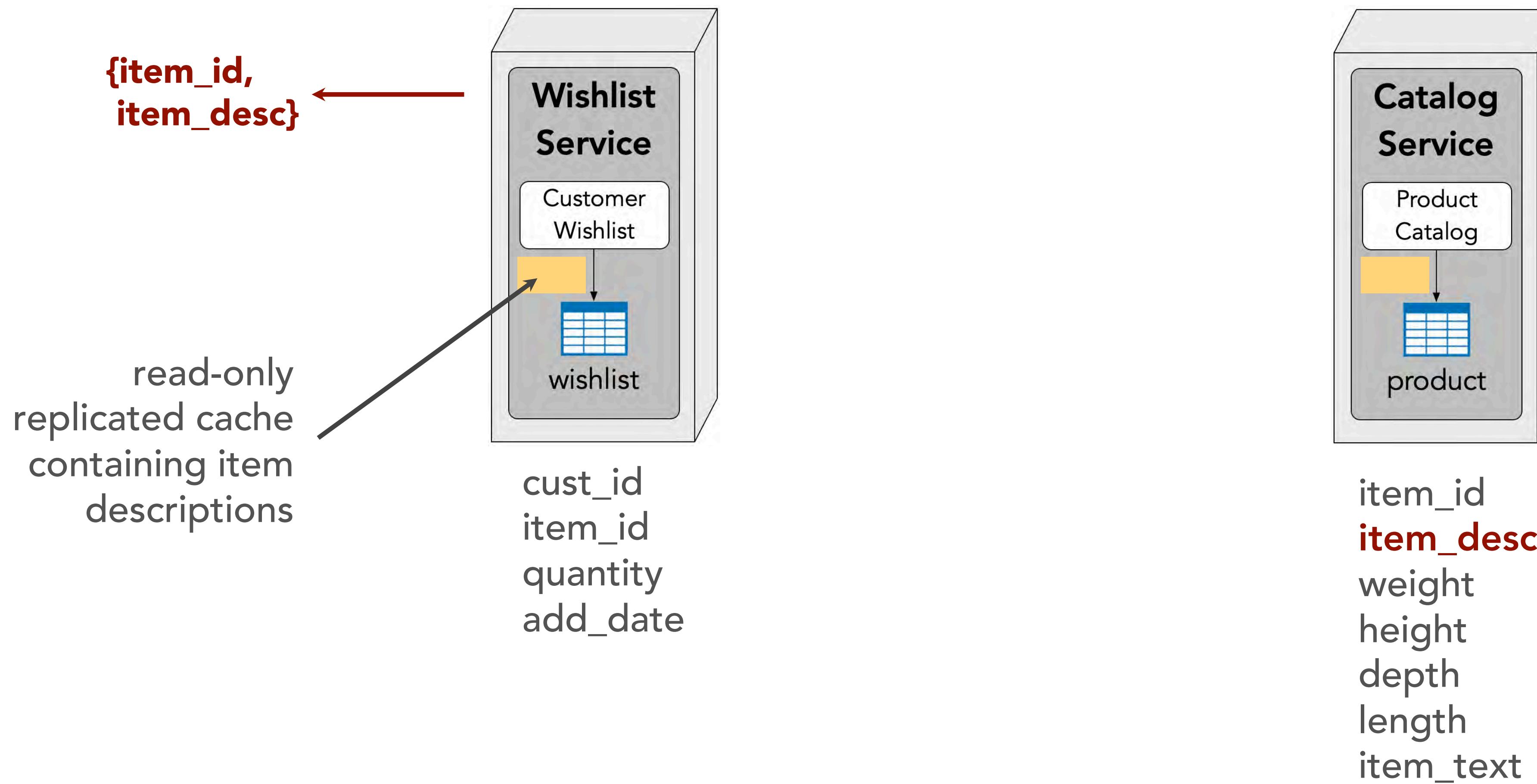
# data access

## option 3: in-memory replicated cache



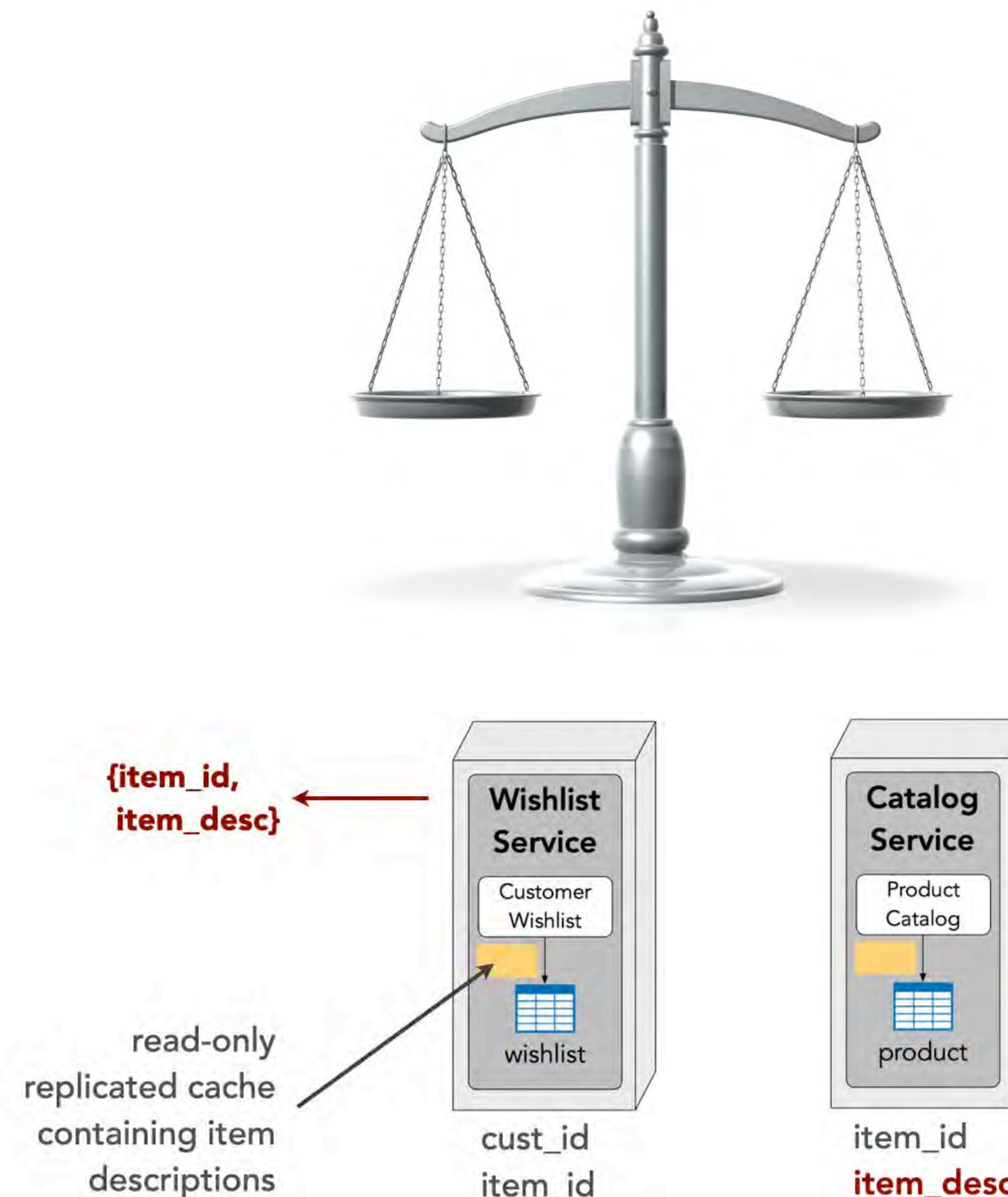
# data access

## option 3: in-memory replicated cache



# tradeoffs

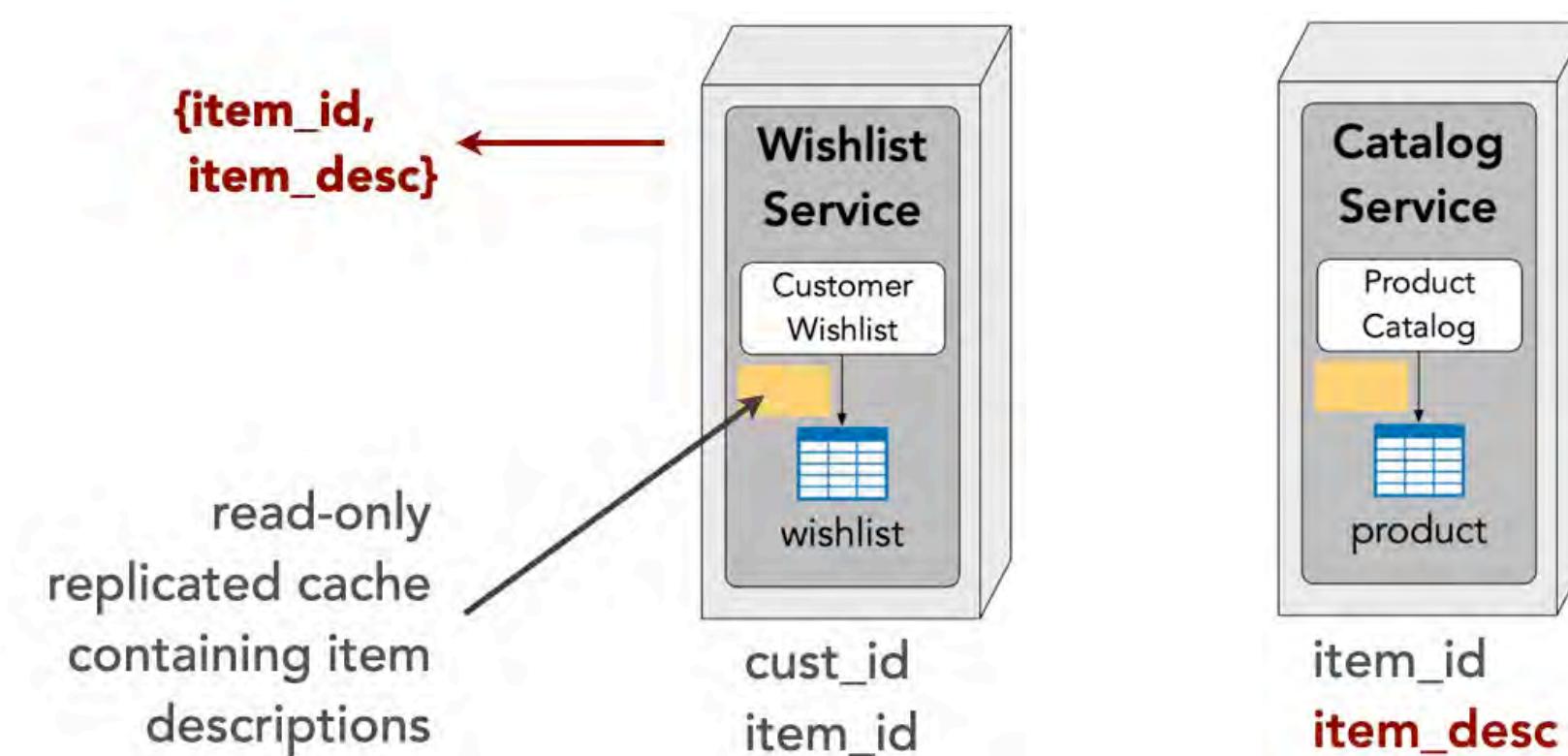
## option 3: in-memory replicated cache



# tradeoffs

## option 3: in-memory replicated cache

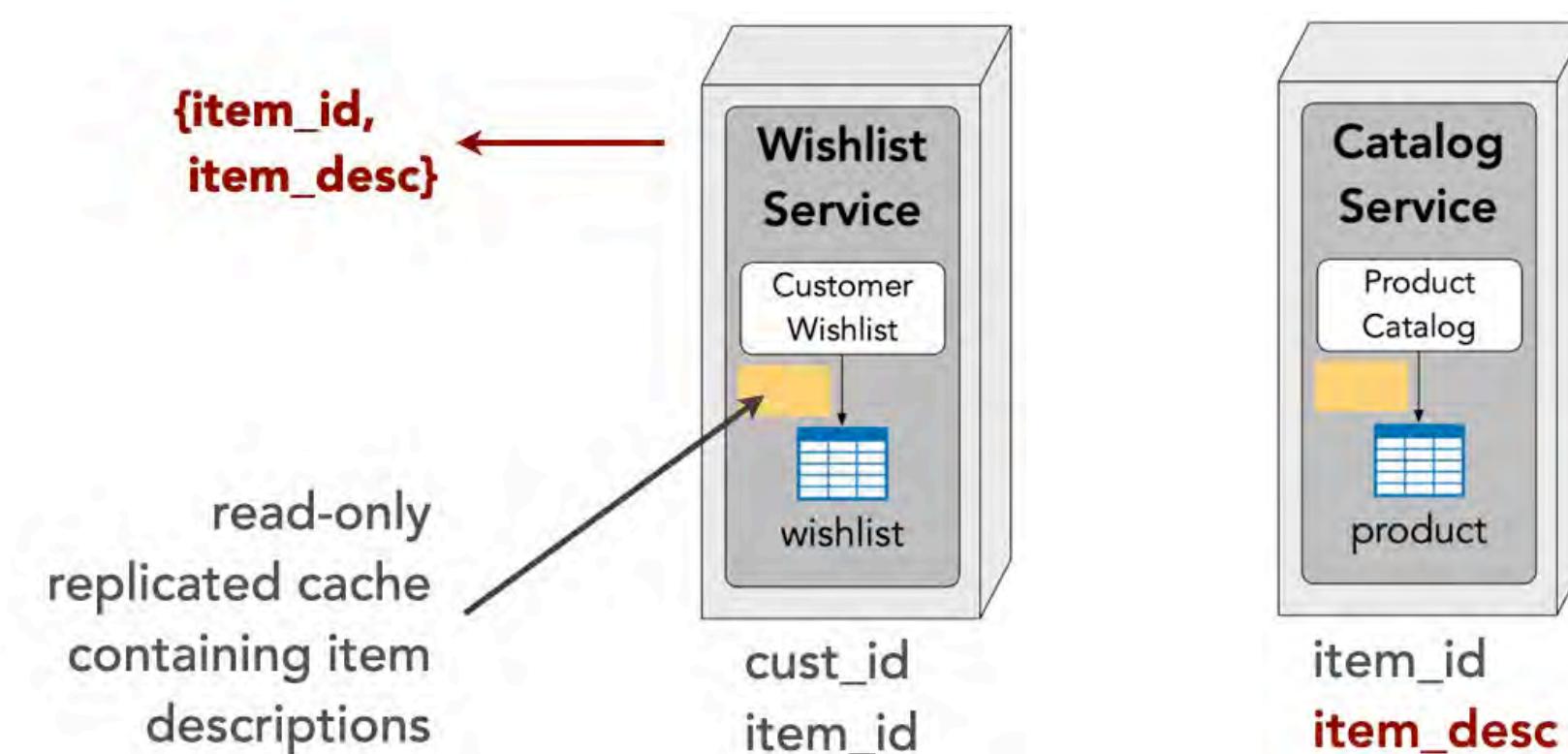
- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance



# tradeoffs

## option 3: in-memory replicated cache

- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance
- ✓ data consistency issues
- ✓ data ownership issues



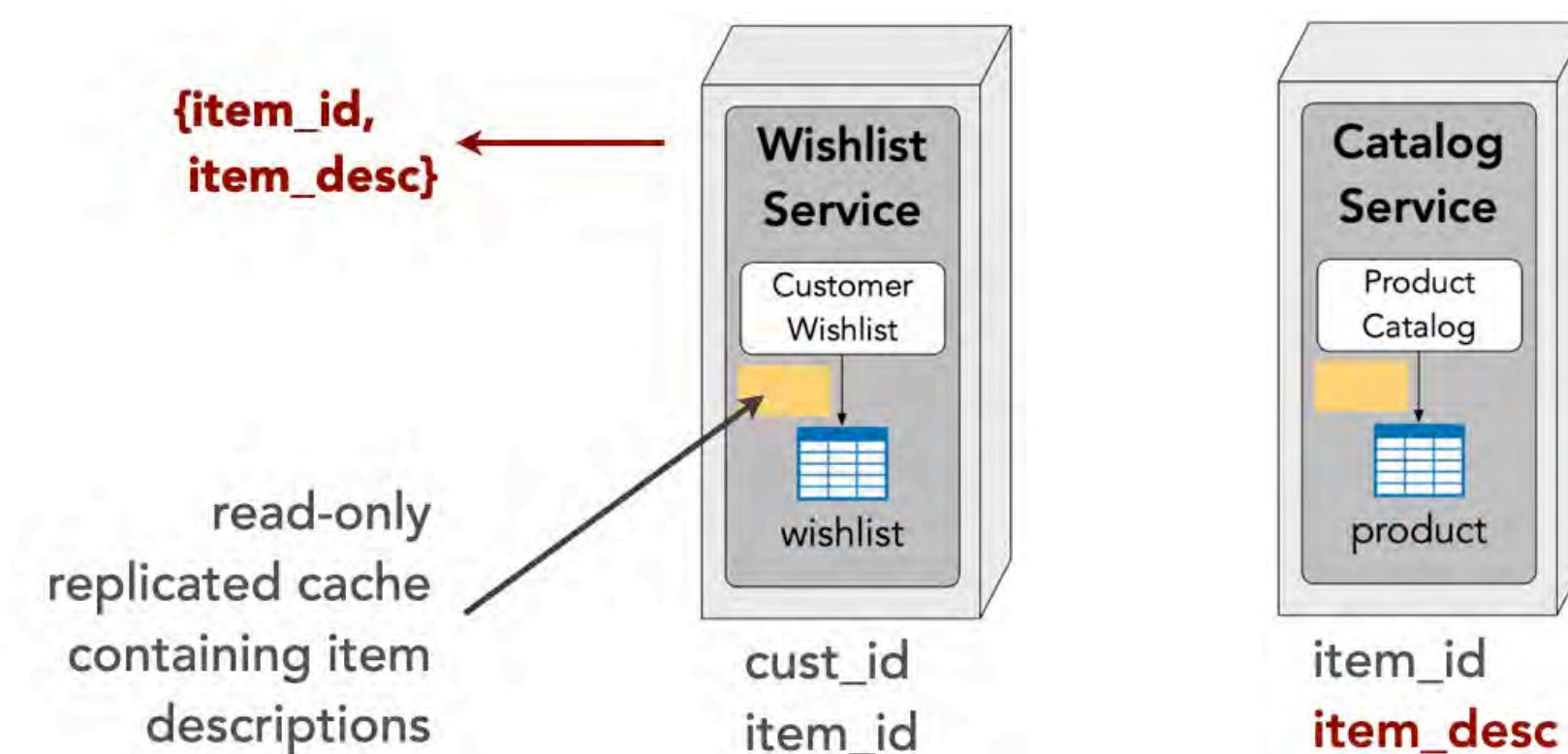
# tradeoffs

## option 3: in-memory replicated cache

- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance
- ✓ data consistency issues
- ✓ data ownership issues

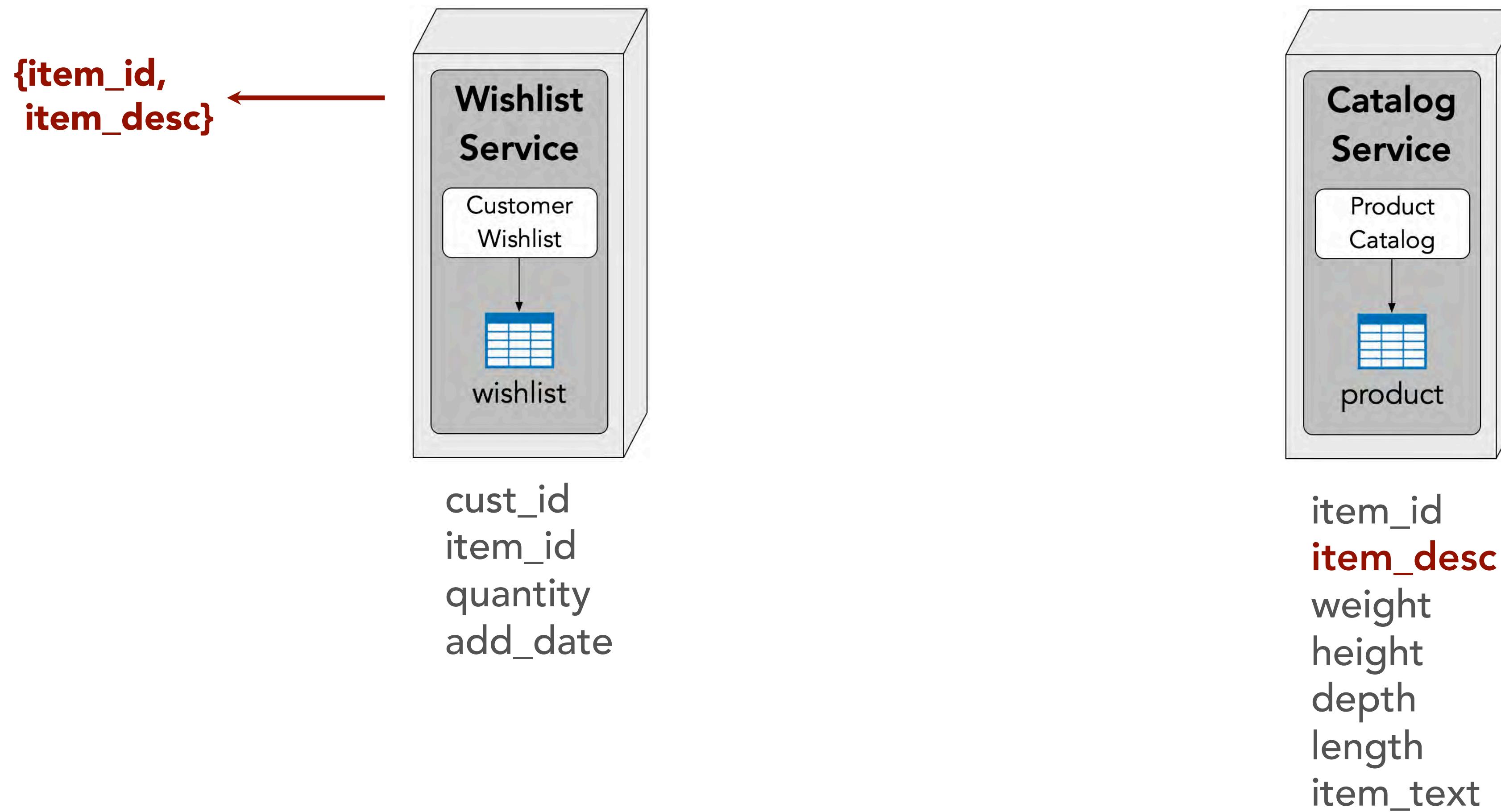


- ! data volume issues
- ! data update rate issues



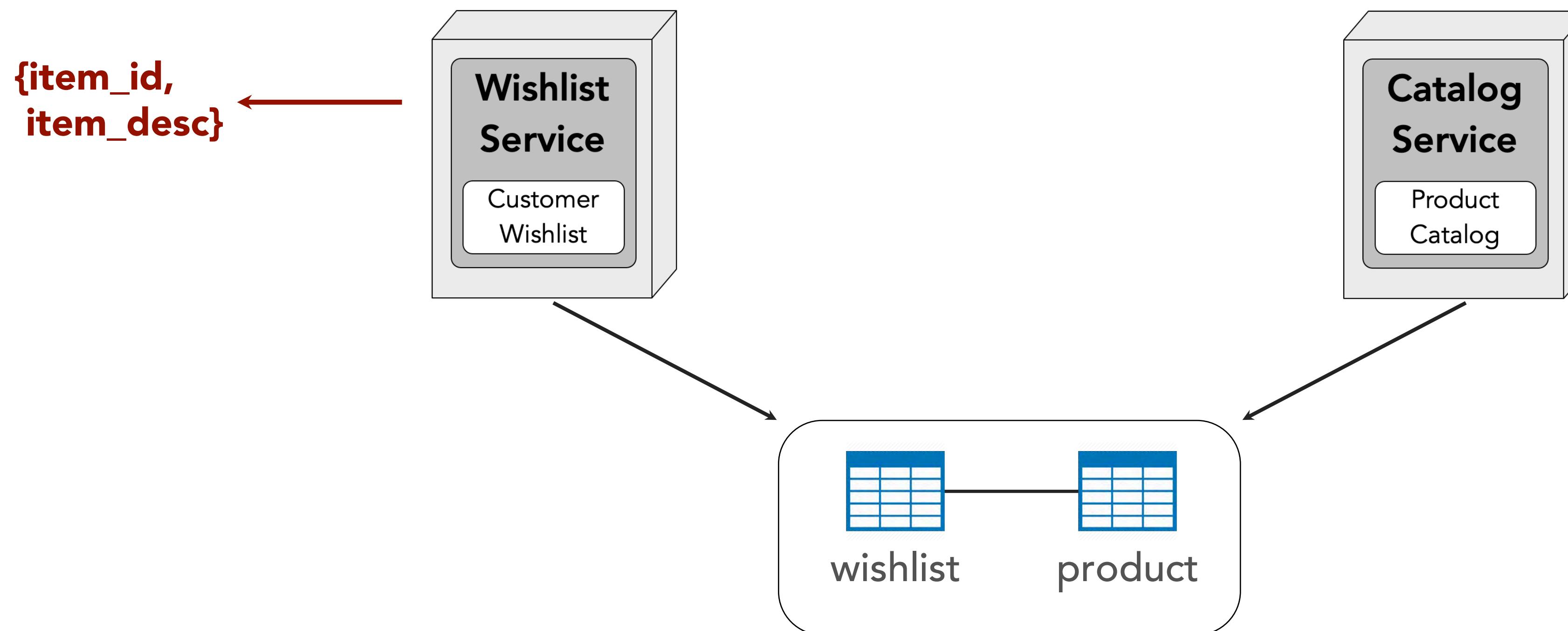
# data access

## option 4: data domain



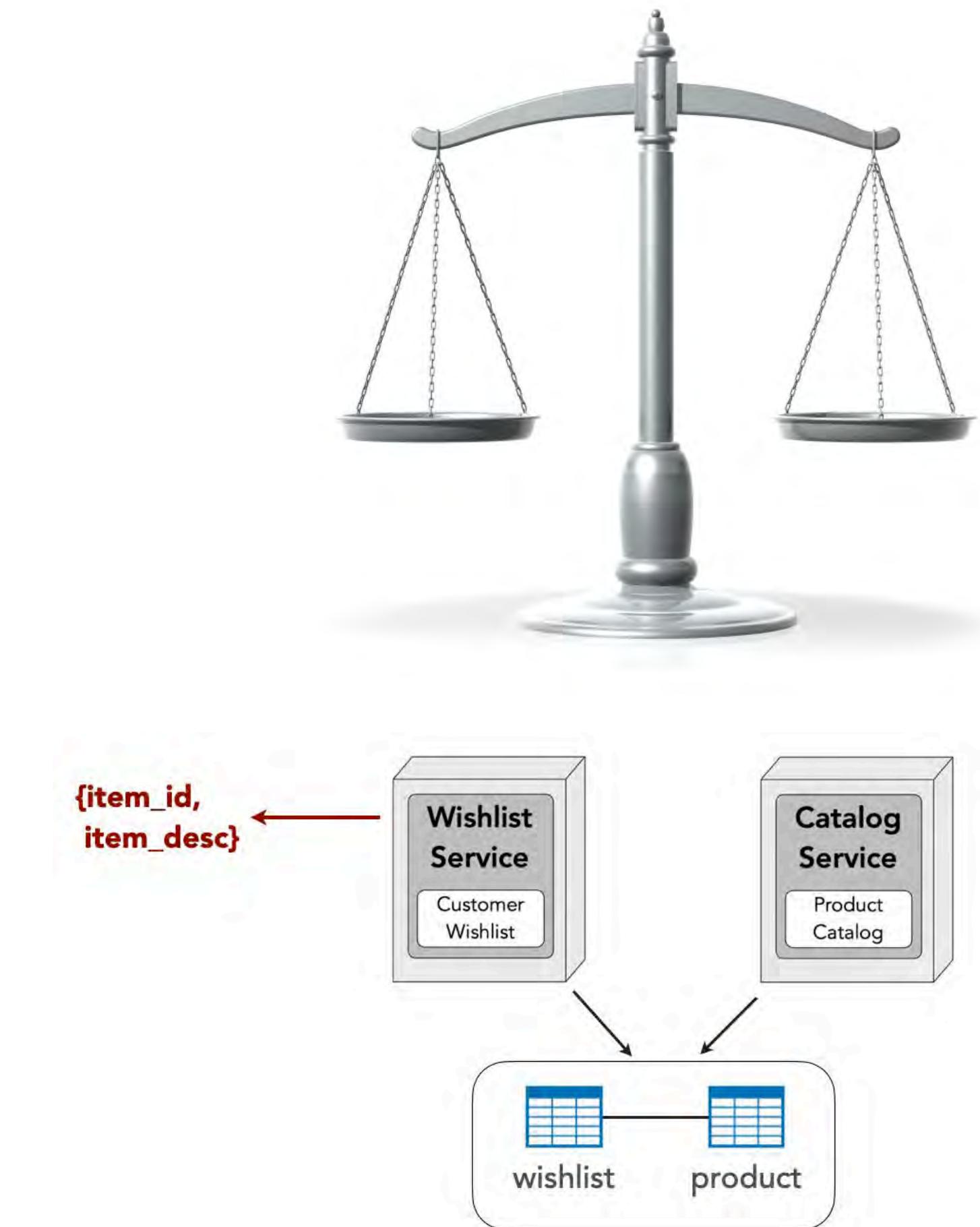
# data access

## option 4: data domain



# tradeoffs

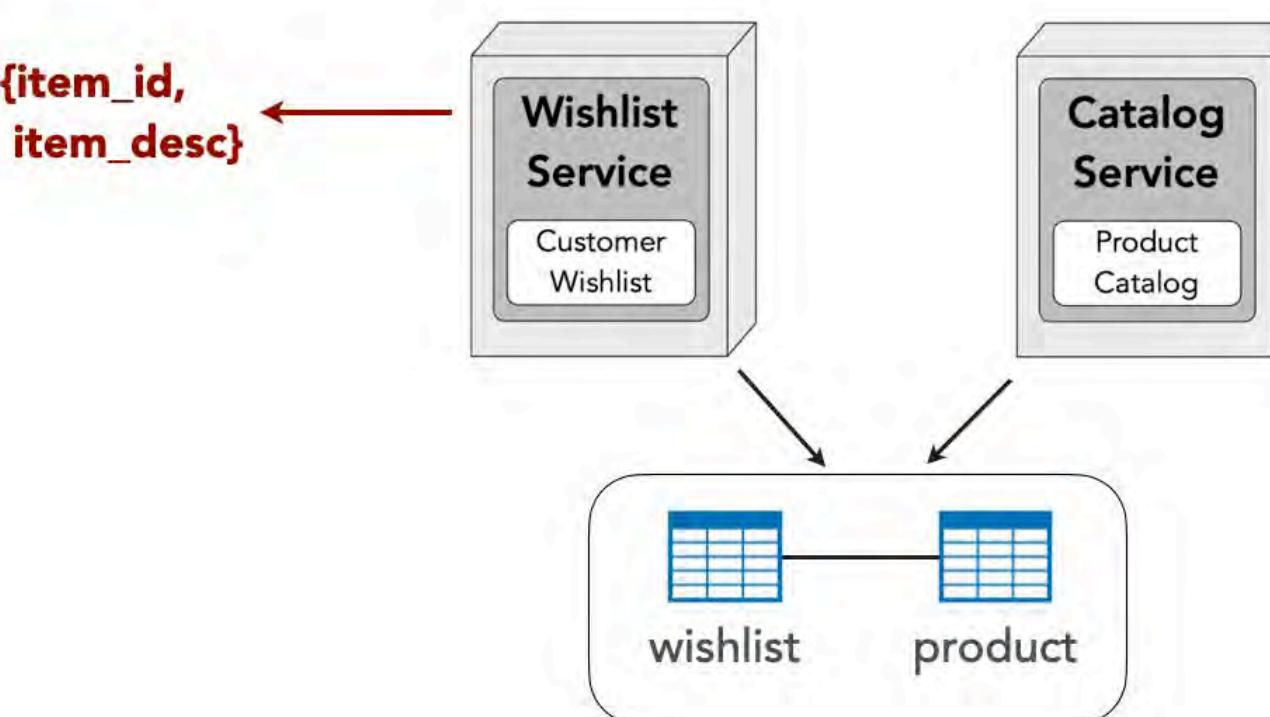
## option 4: data domain



# tradeoffs

## option 4: data domain

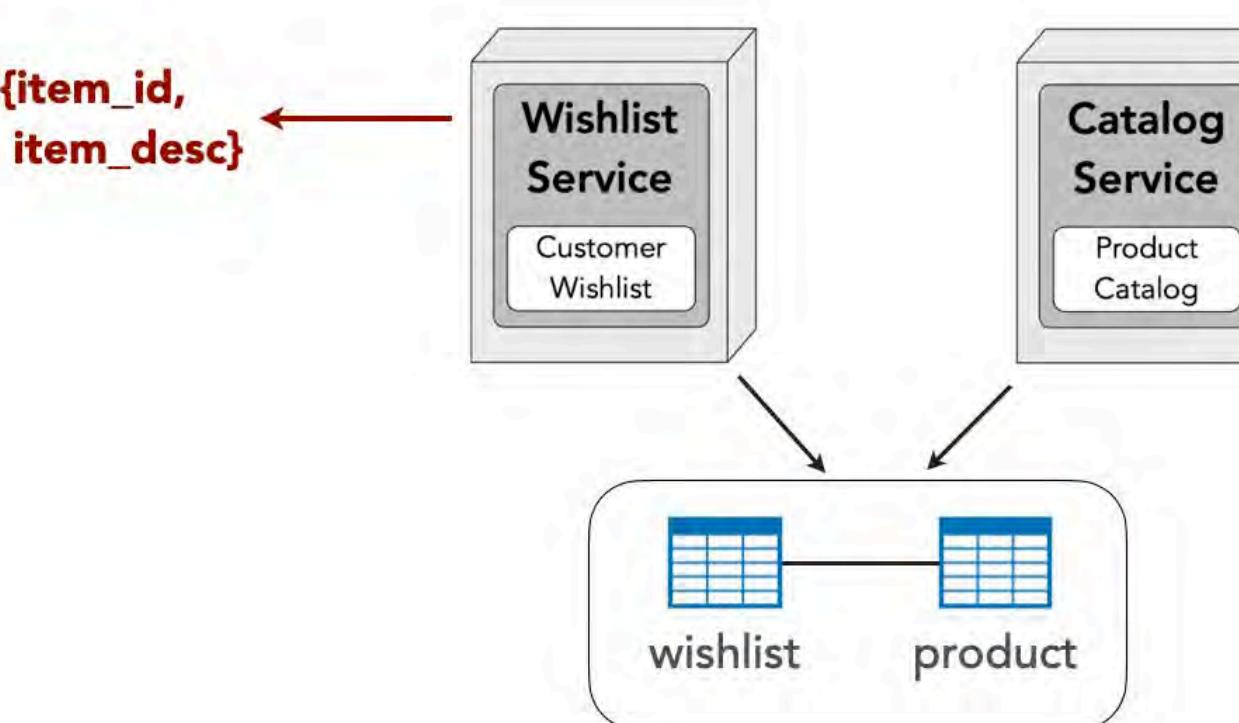
- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance



# tradeoffs

## option 4: data domain

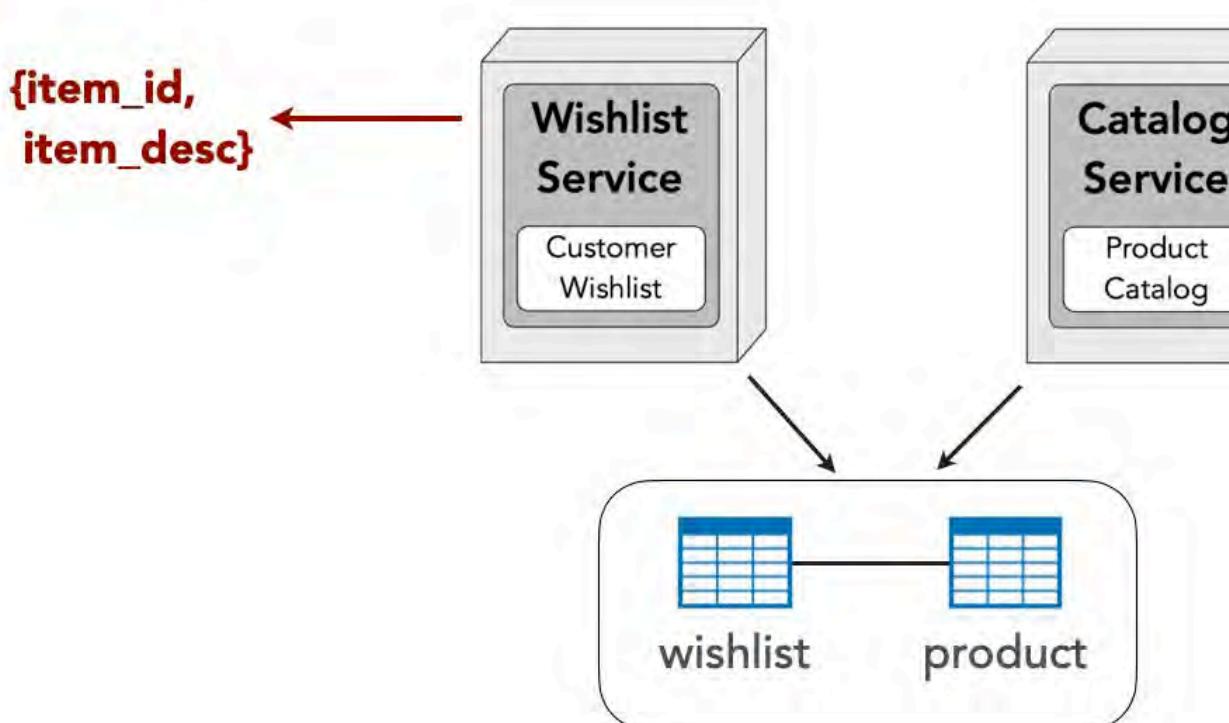
- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance
- ✓ data consistency issues
- ✓ data ownership issues



# tradeoffs

## option 4: data domain

- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance
- ✓ data consistency issues
- ✓ data ownership issues
- ✓ data volume issues
- ✓ data update rate issues



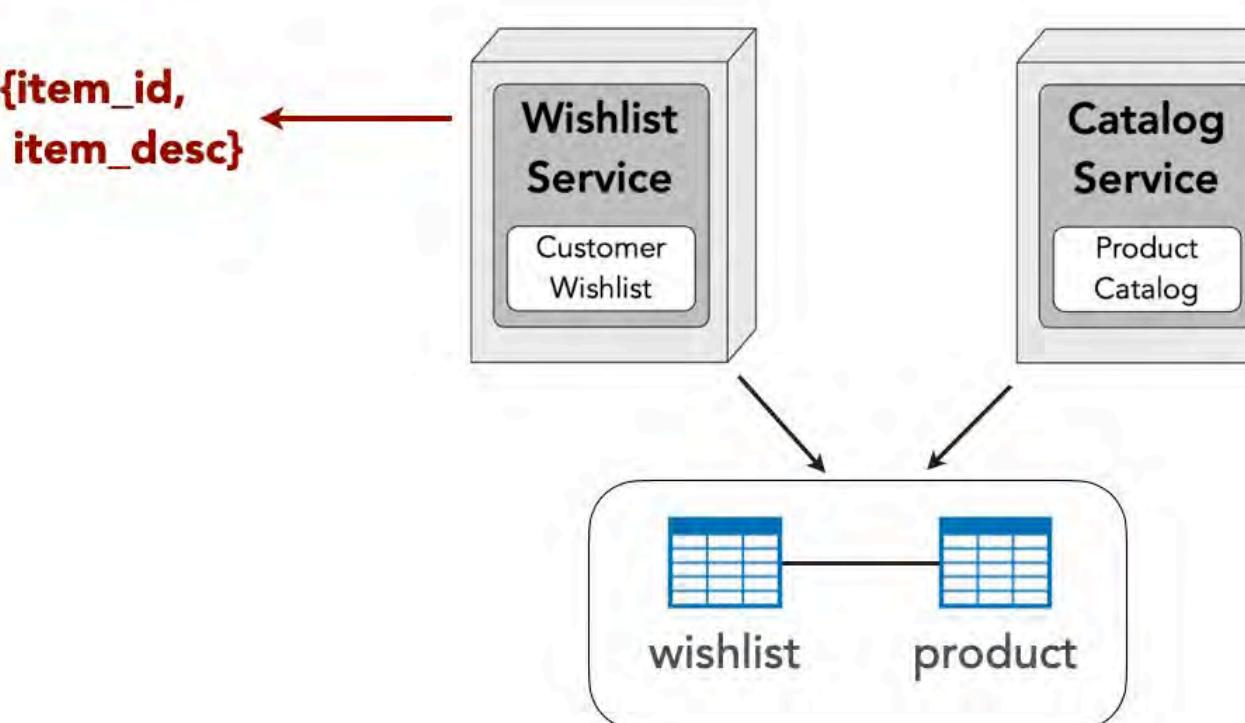
# tradeoffs

## option 4: data domain

- ✓ network and security latency
- ✓ scalability and throughput
- ✓ fault tolerance
- ✓ data consistency issues
- ✓ data ownership issues
- ✓ data volume issues
- ✓ data update rate issues



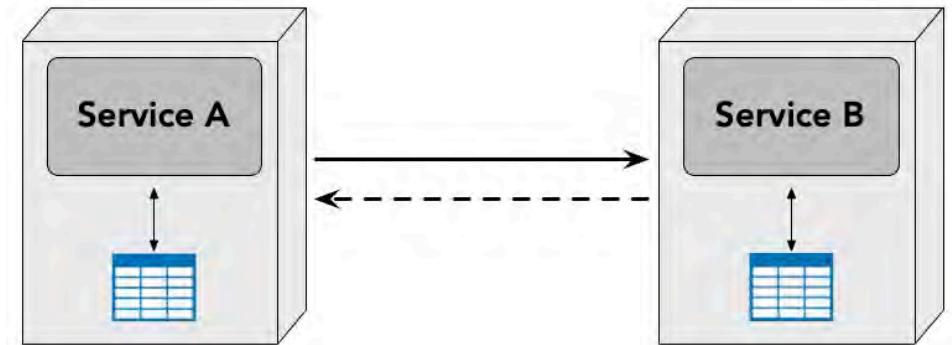
- ! change control issues
- ! read/write responsibilities
- ! possible security issues
- ! broader bounded context



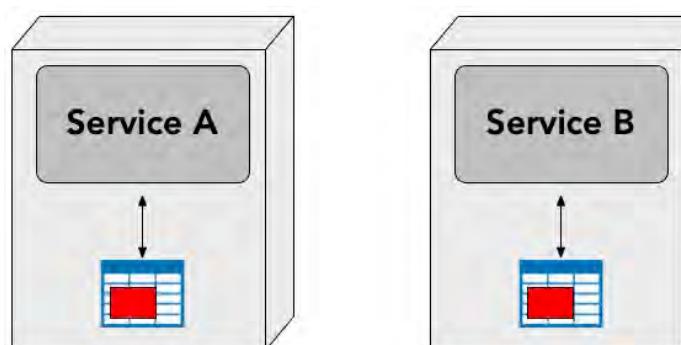
# data access

which one is better?

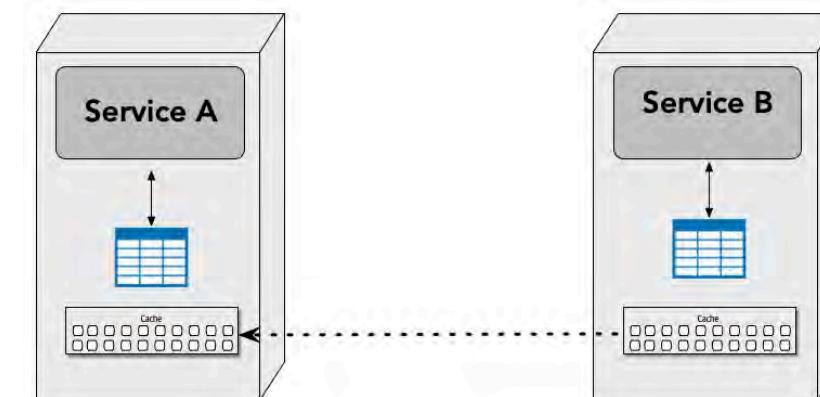
option 1:  
interservice  
communication



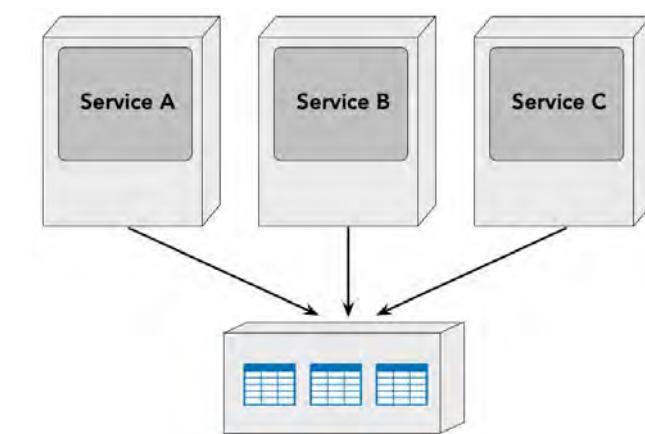
option 2:  
data schema  
replication



option 3:  
in-memory  
replicated cache



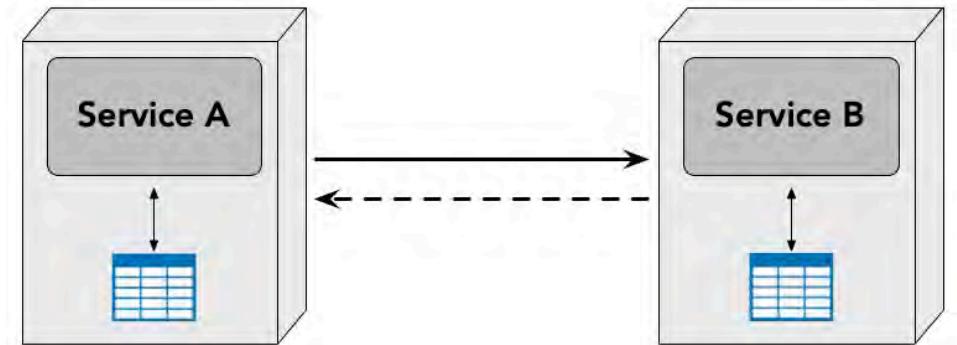
option 4:  
data domains  
(shared tables)



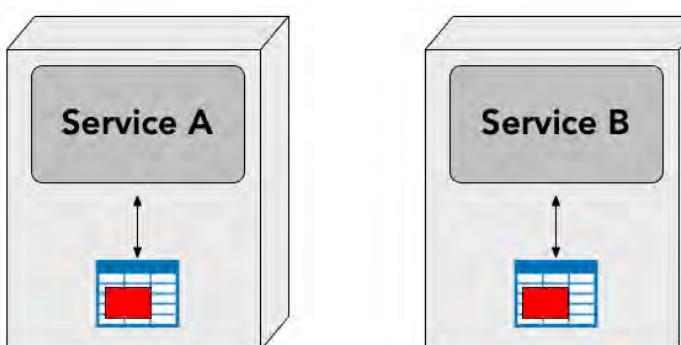
# data access

which one is better?

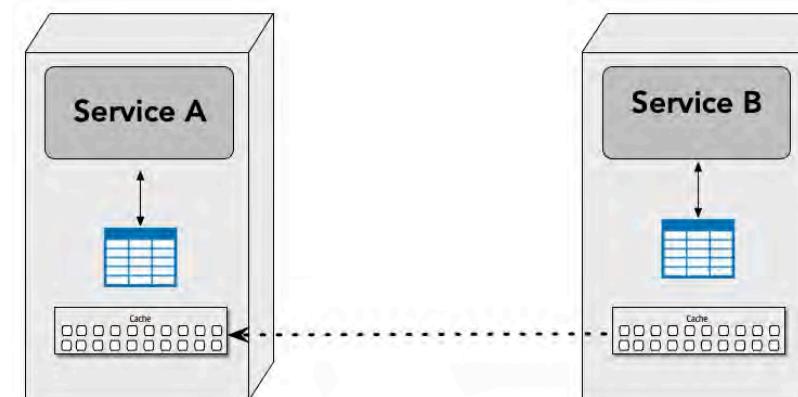
option 1:  
interservice  
communication



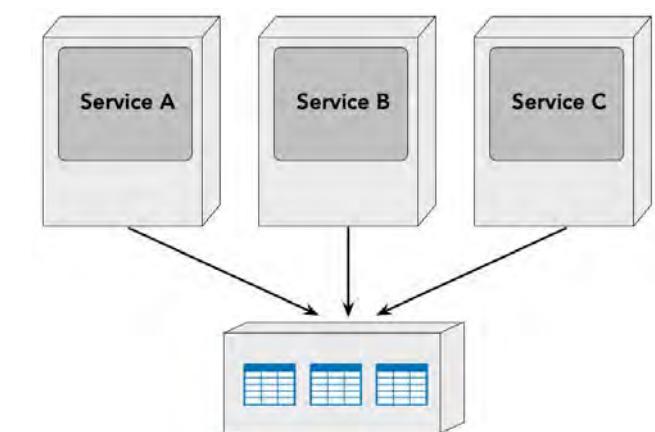
option 2:  
data schema  
replication



option 3:  
in-memory  
replicated cache



option 4:  
data domains  
(shared tables)

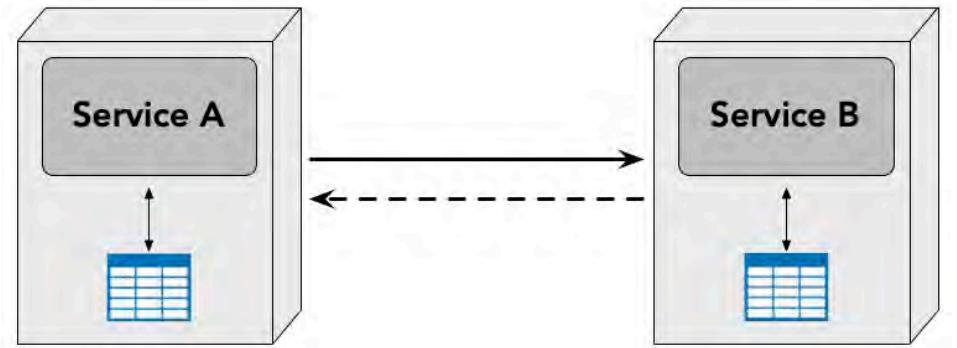


- ✓ large data volume
- ✓ low responsiveness

# data access

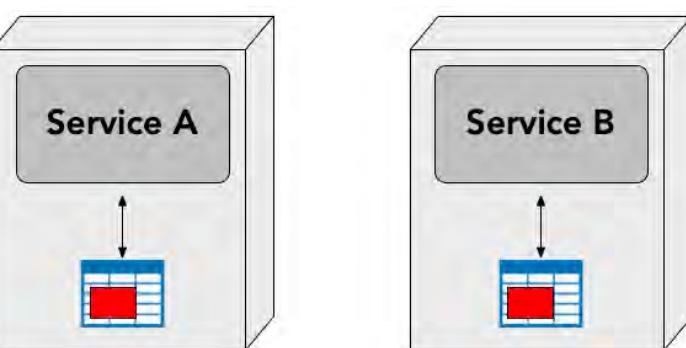
which one is better?

option 1:  
interservice  
communication



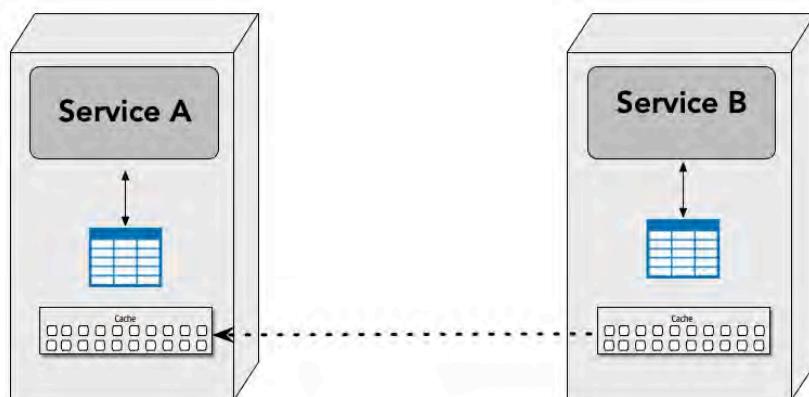
- ✓ large data volume
- ✓ low responsiveness

option 2:  
data schema  
replication

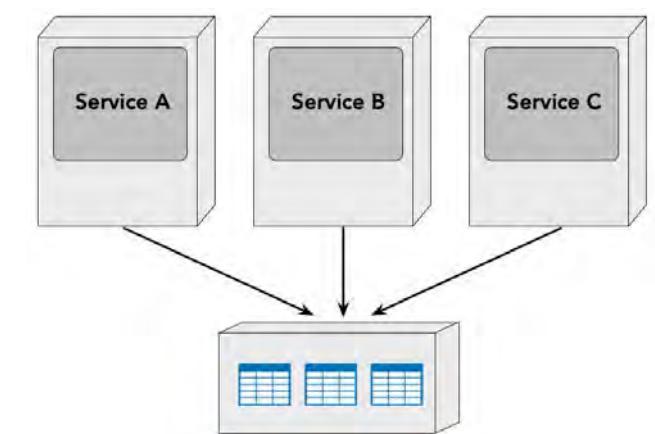


- ✓ reporting
- ✓ data aggregation

option 3:  
in-memory  
replicated cache



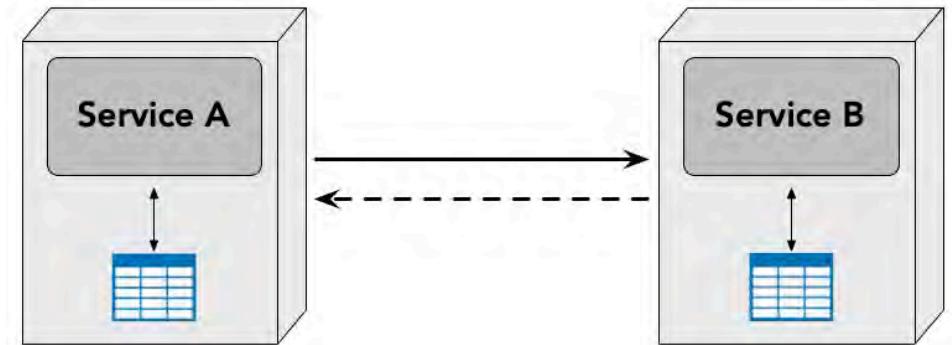
option 4:  
data domains  
(shared tables)



# data access

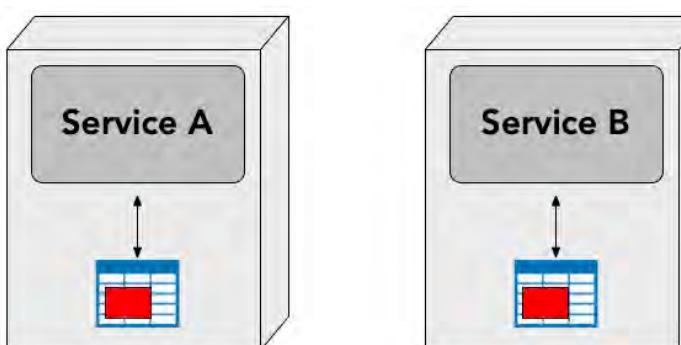
which one is better?

option 1:  
interservice  
communication



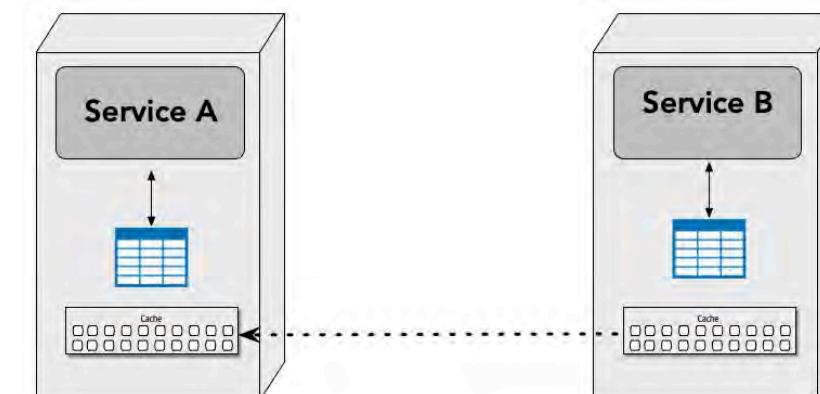
- ✓ large data volume
- ✓ low responsiveness

option 2:  
data schema  
replication



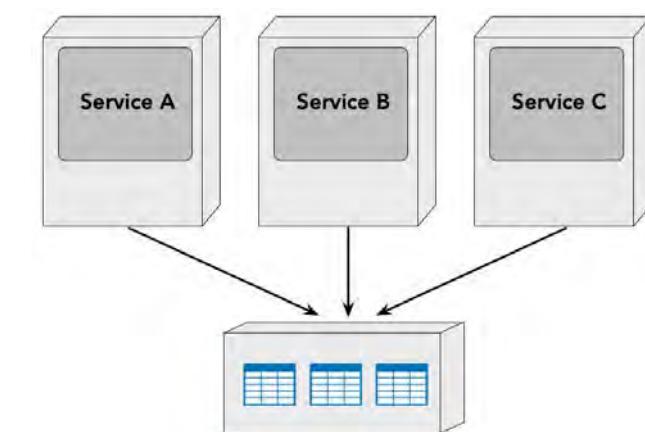
- ✓ reporting
- ✓ data aggregation

option 3:  
in-memory  
replicated cache



- ✓ low data volume
- ✓ high responsiveness

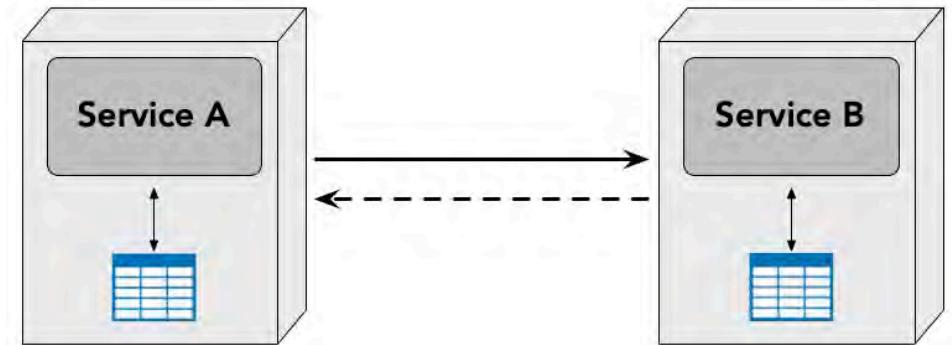
option 4:  
data domains  
(shared tables)



# data access

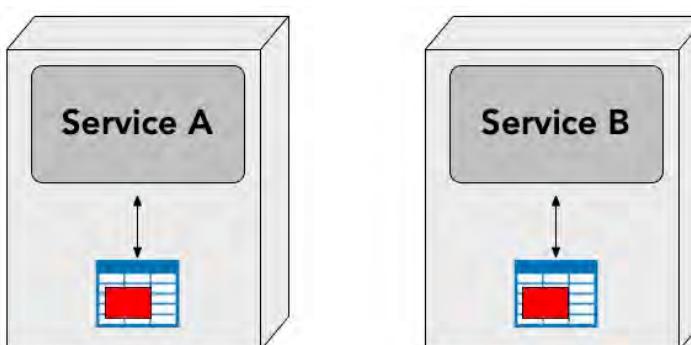
which one is better?

option 1:  
interservice  
communication



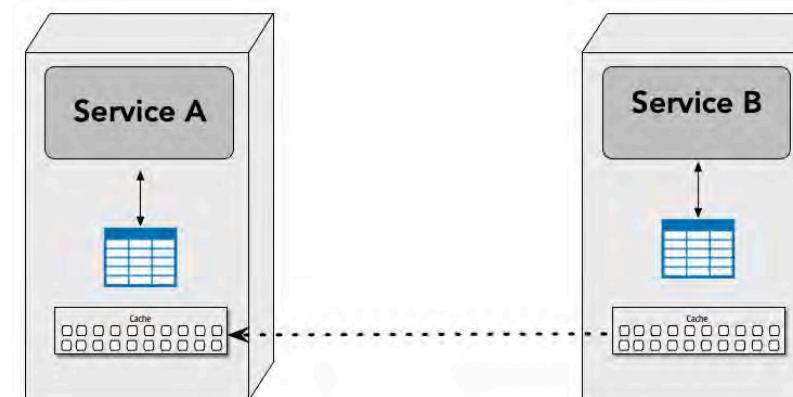
- ✓ large data volume
- ✓ low responsiveness

option 2:  
data schema  
replication



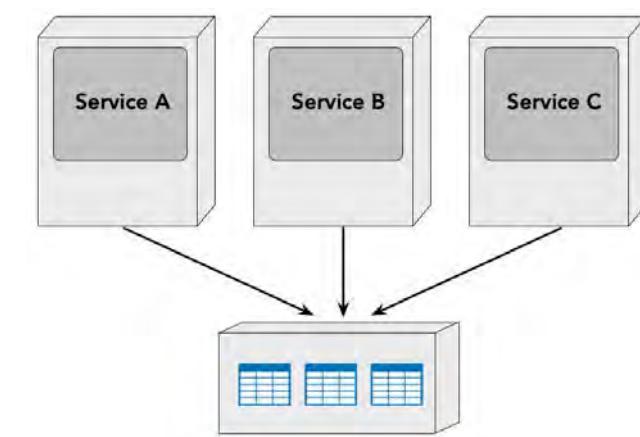
- ✓ reporting
- ✓ data aggregation

option 3:  
in-memory  
replicated cache



- ✓ low data volume
- ✓ high responsiveness

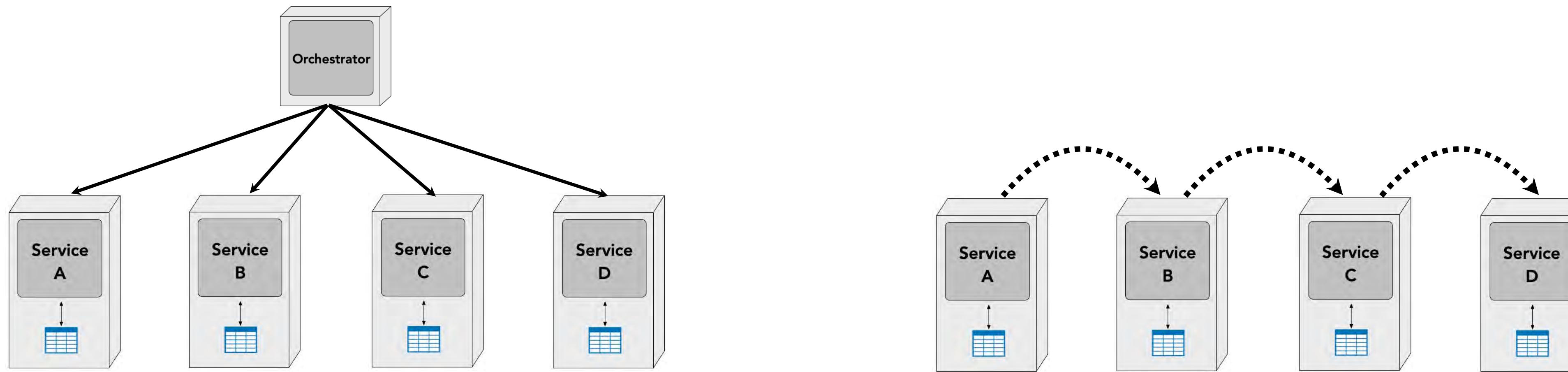
option 4:  
data domains  
(shared tables)



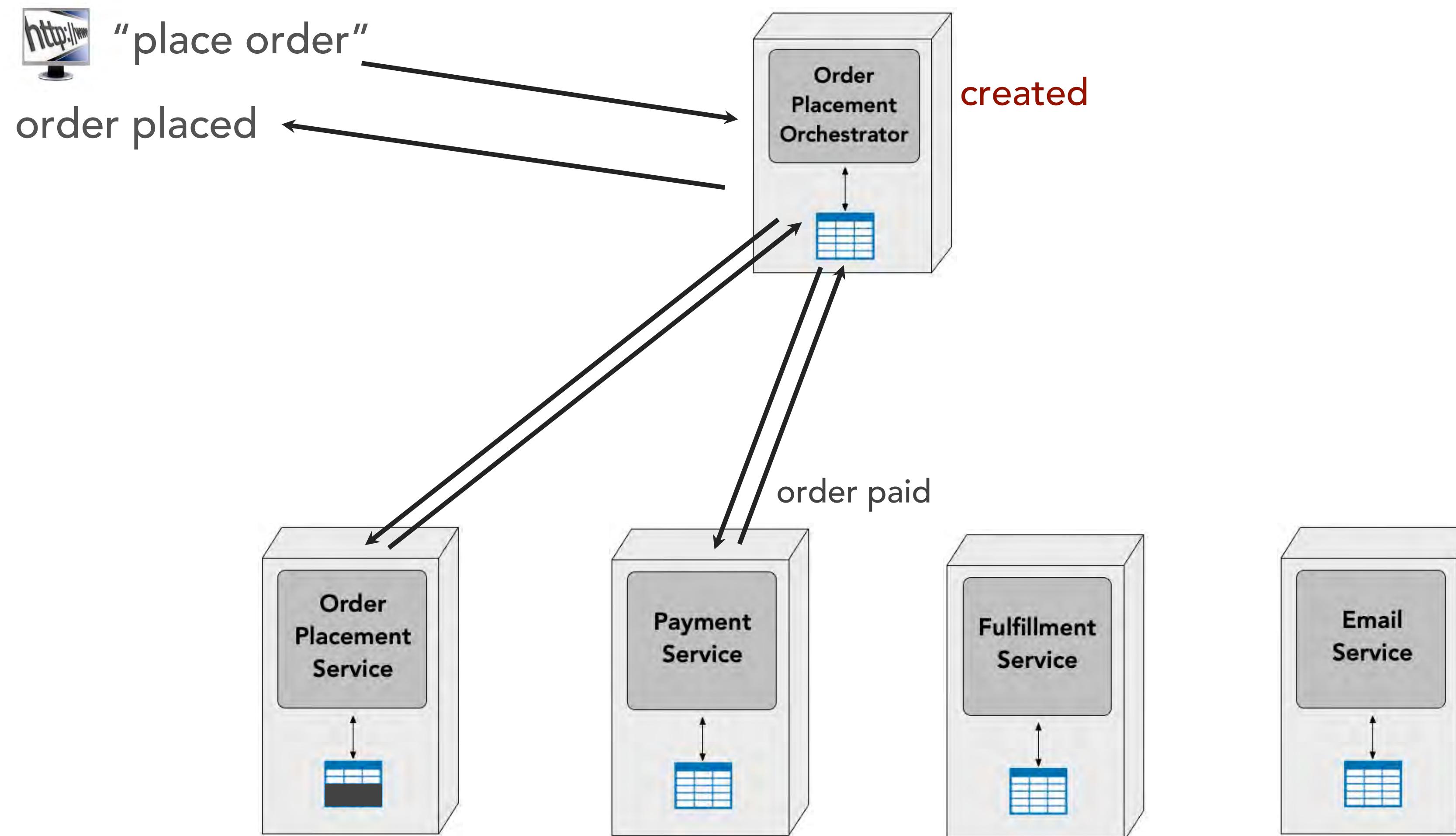
- ✓ multiple owners (write)
- ✓ high responsiveness

# Managing Workflows

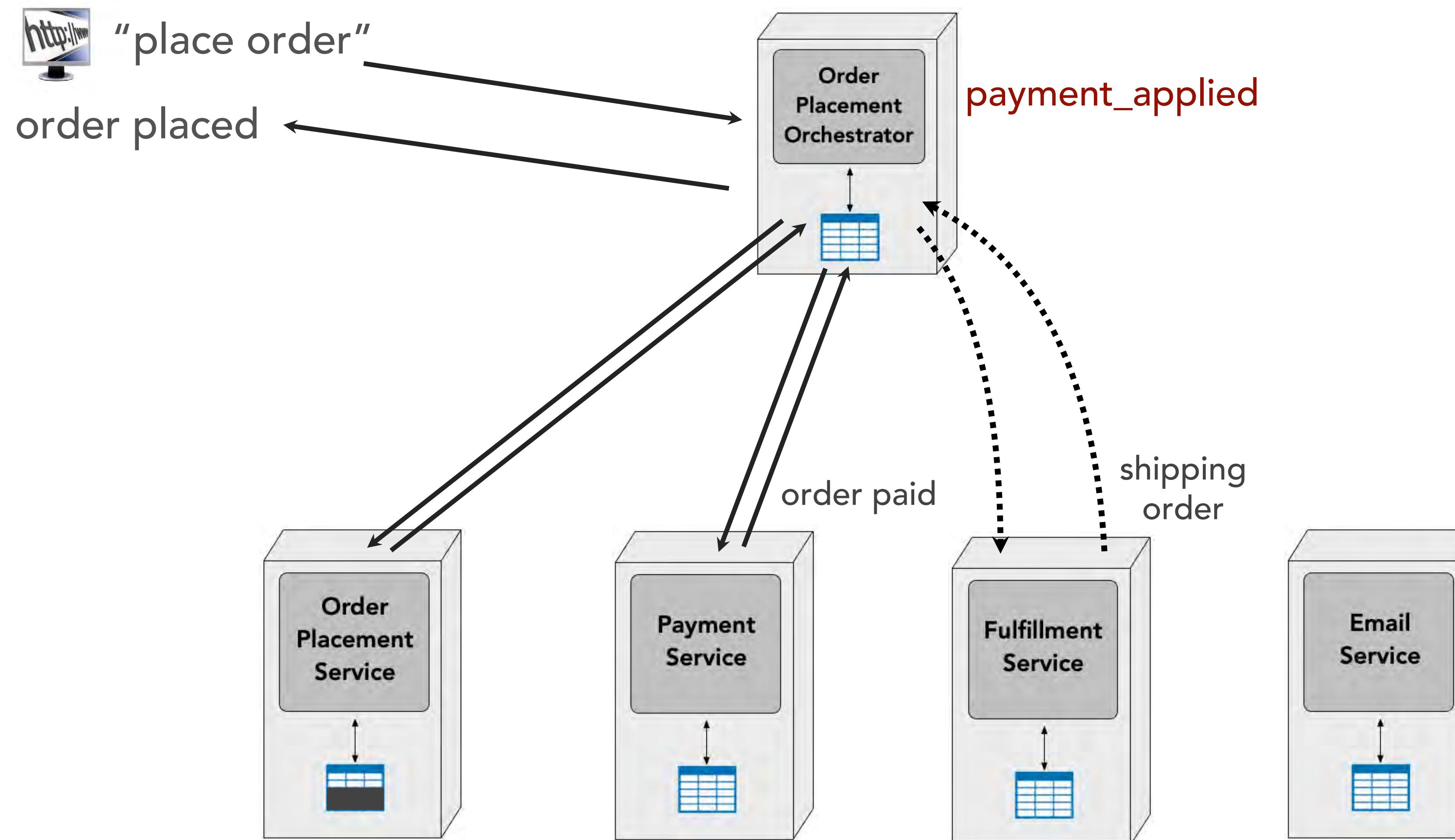
# managing workflows



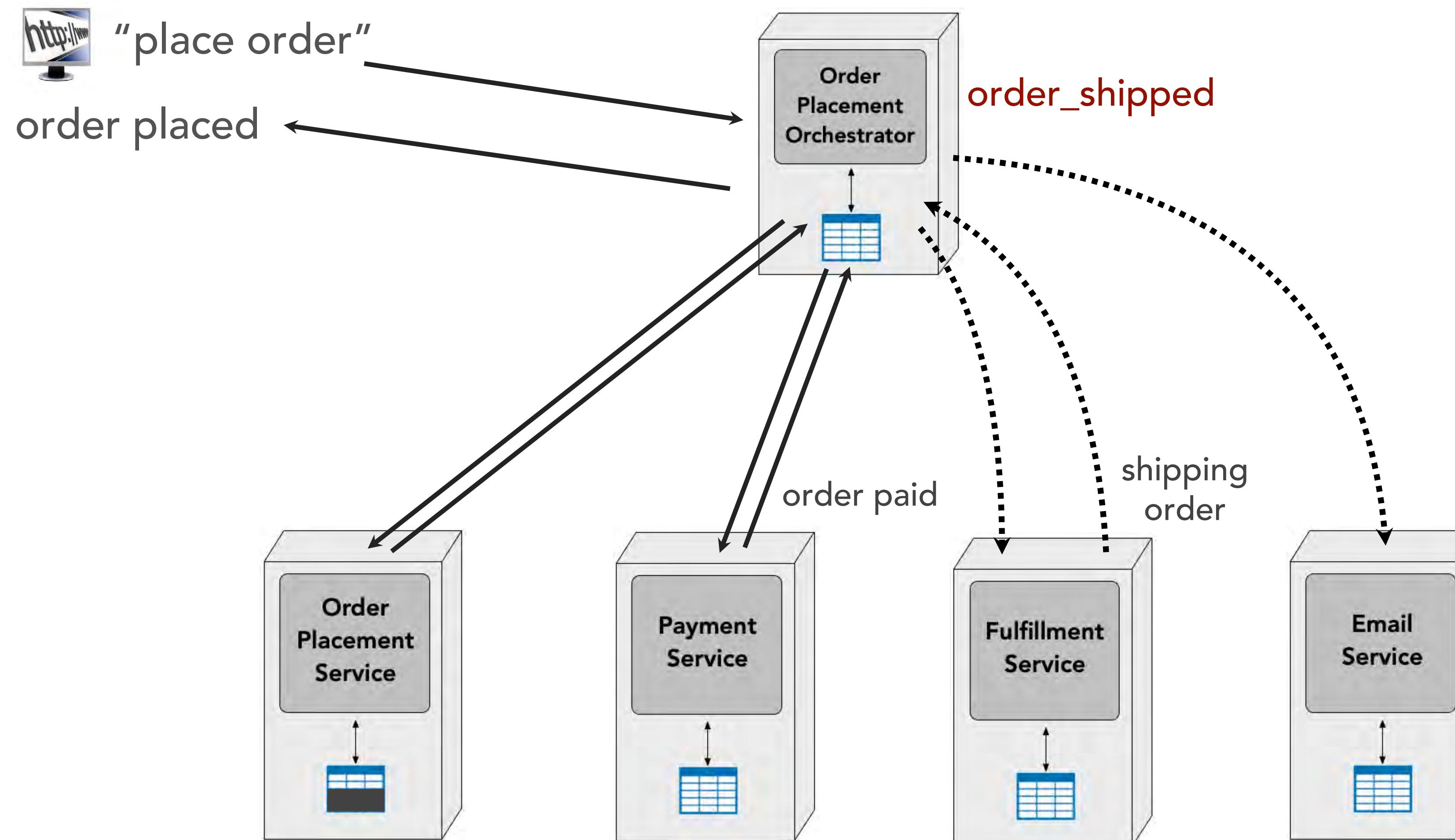
# managing workflows



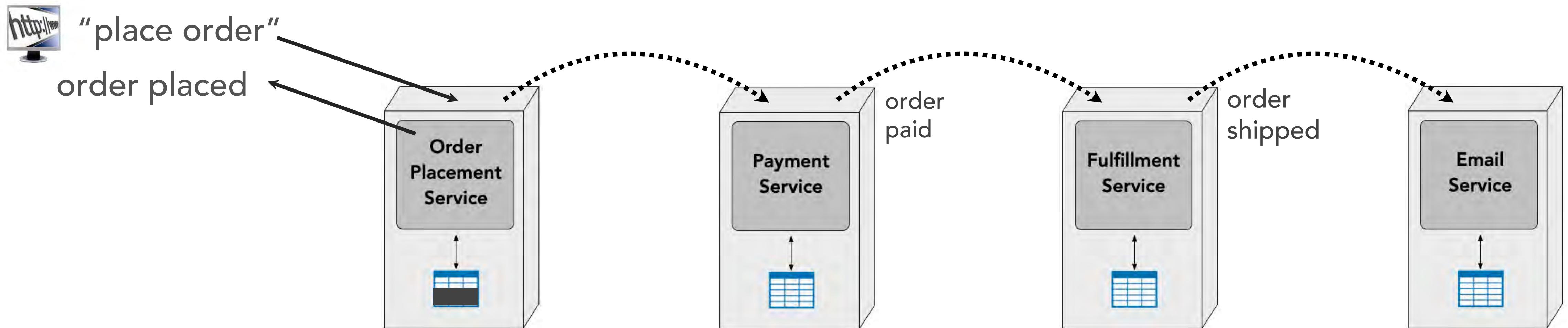
# managing workflows



# managing workflows

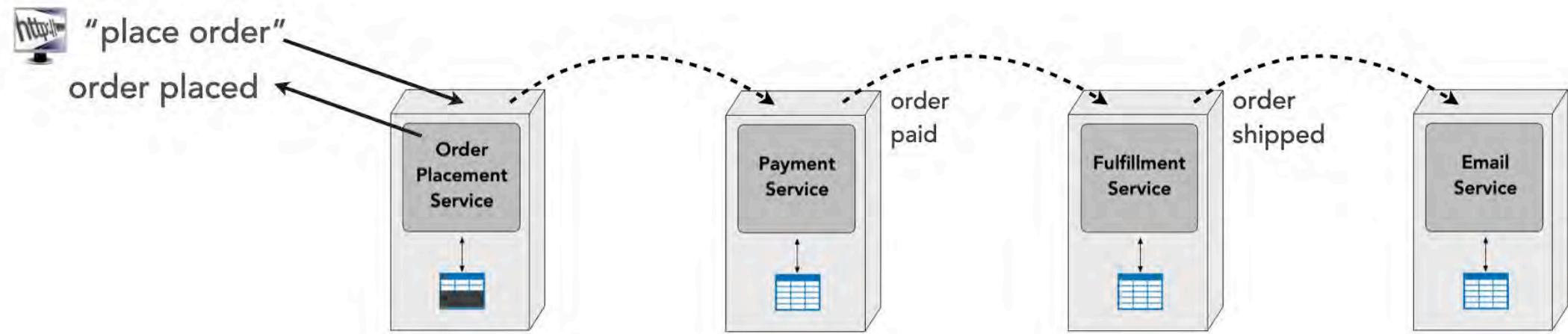


# managing workflows

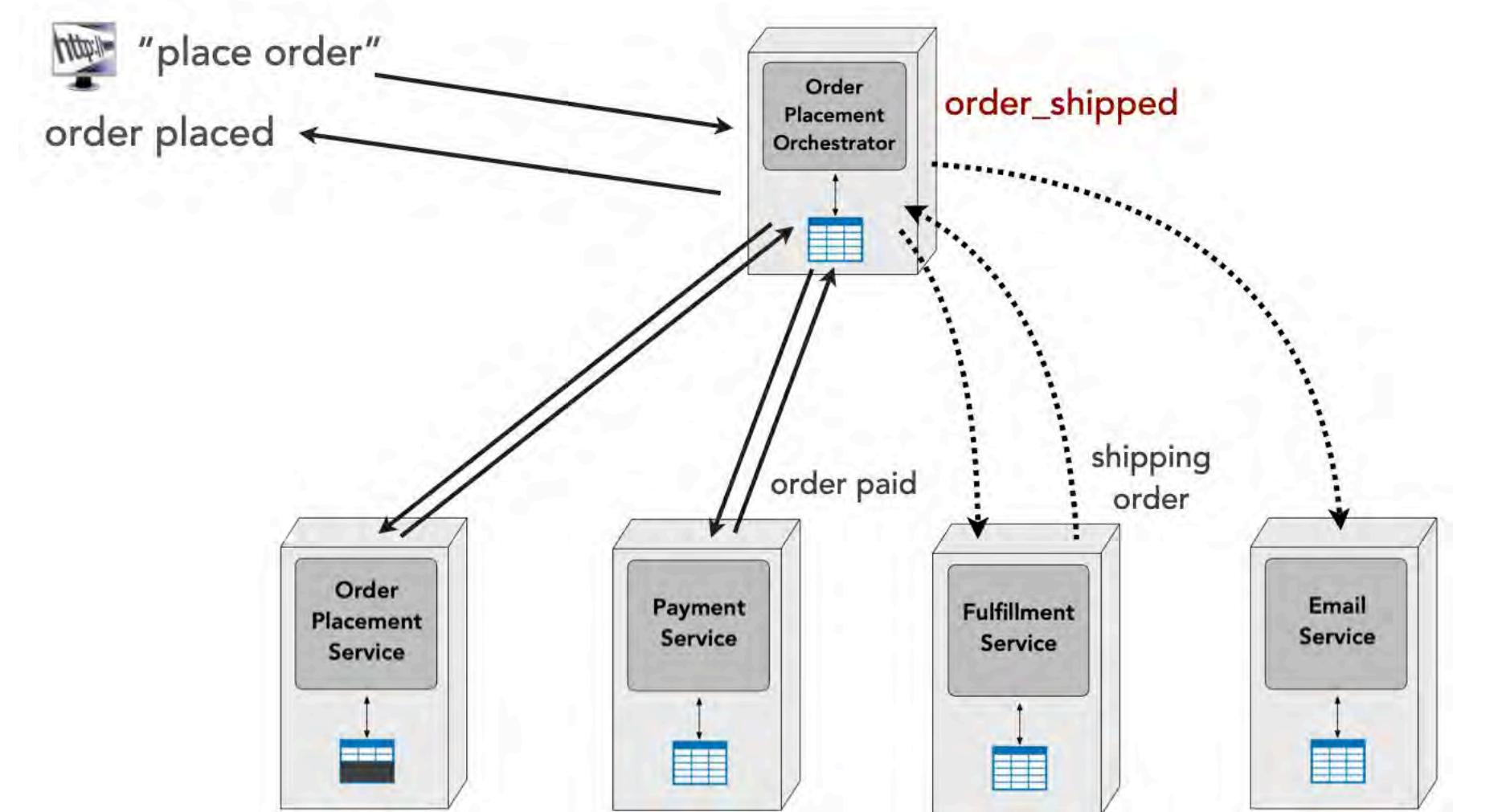


# managing workflows

choreography

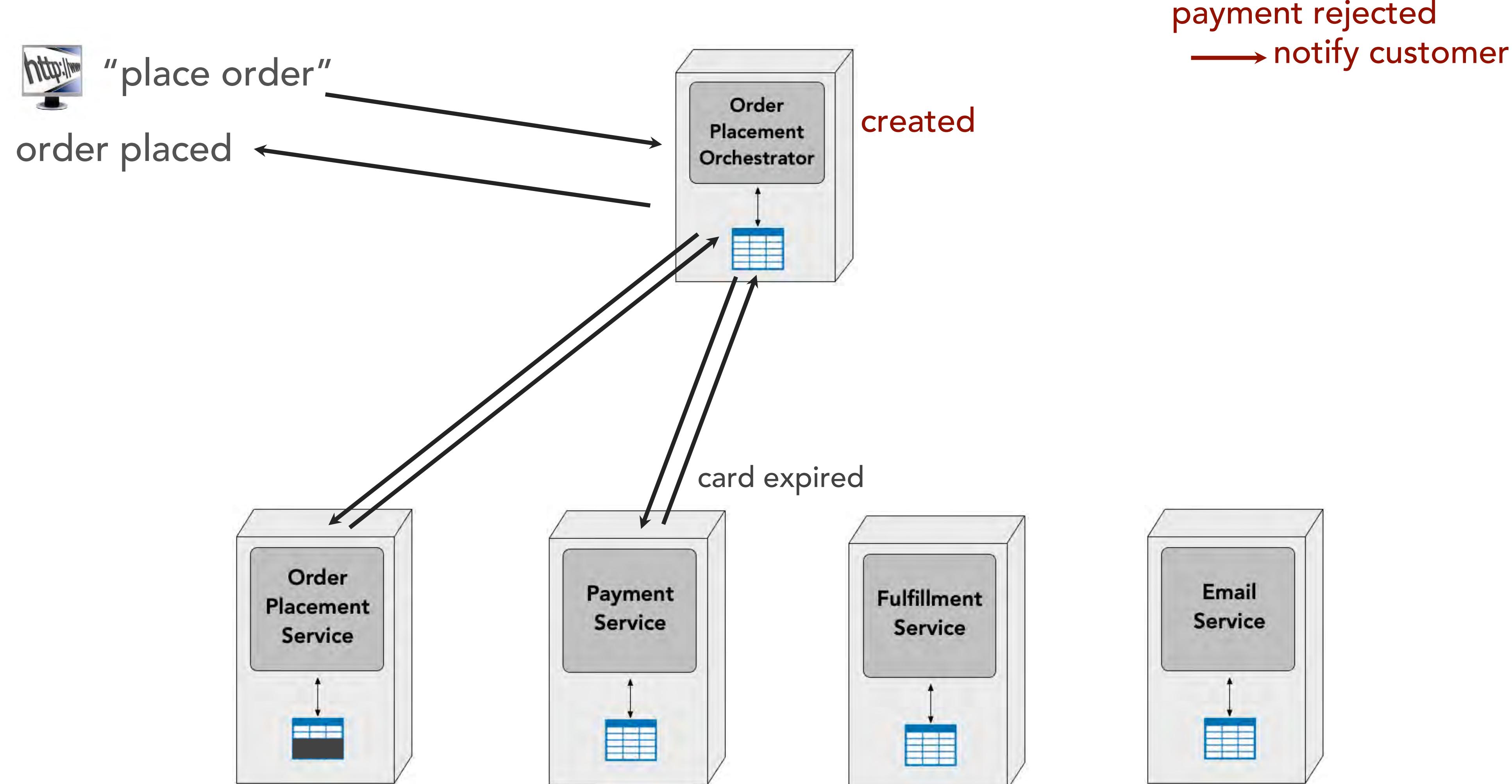


orchestration



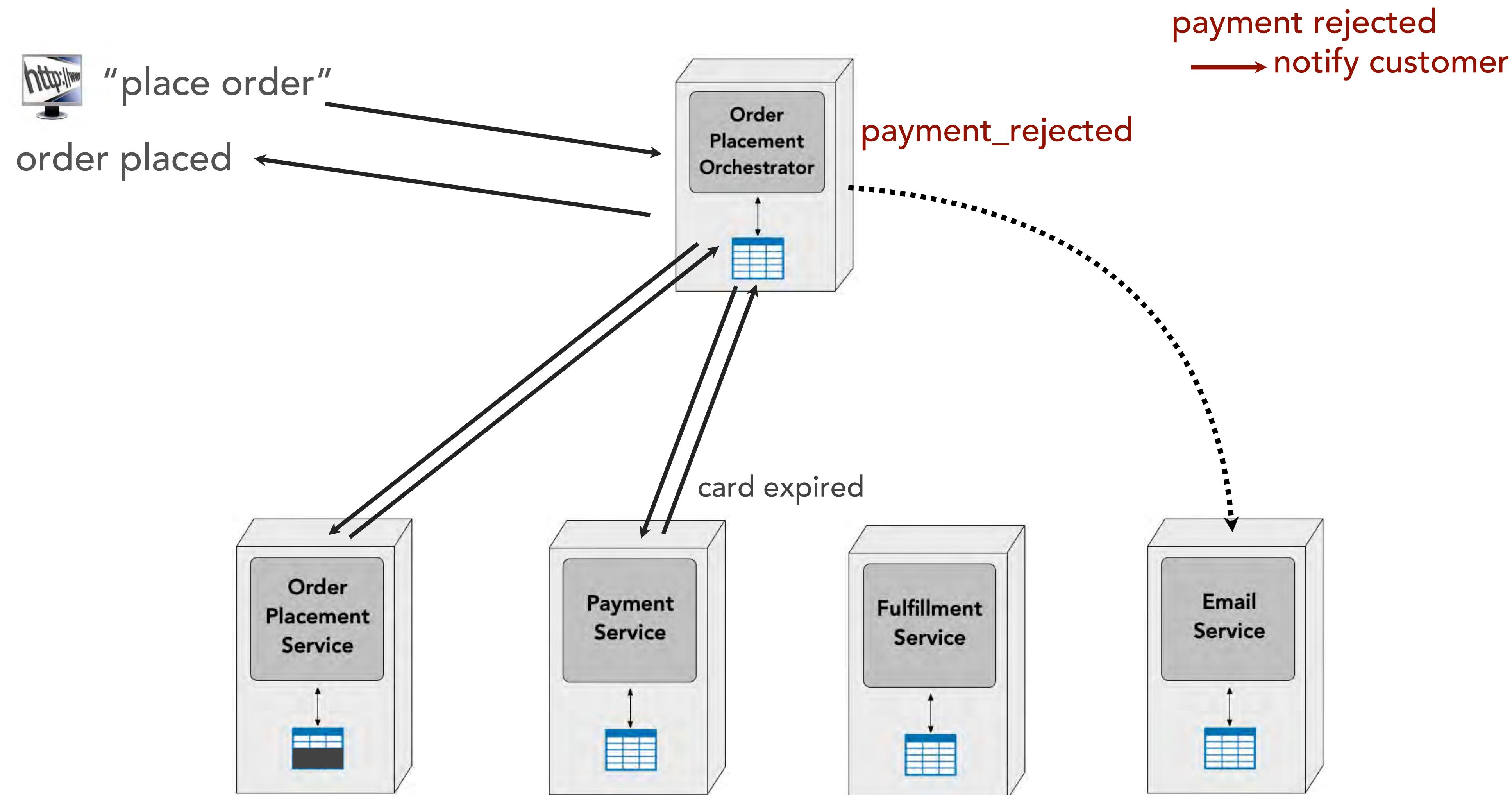
# managing workflows

## error handling



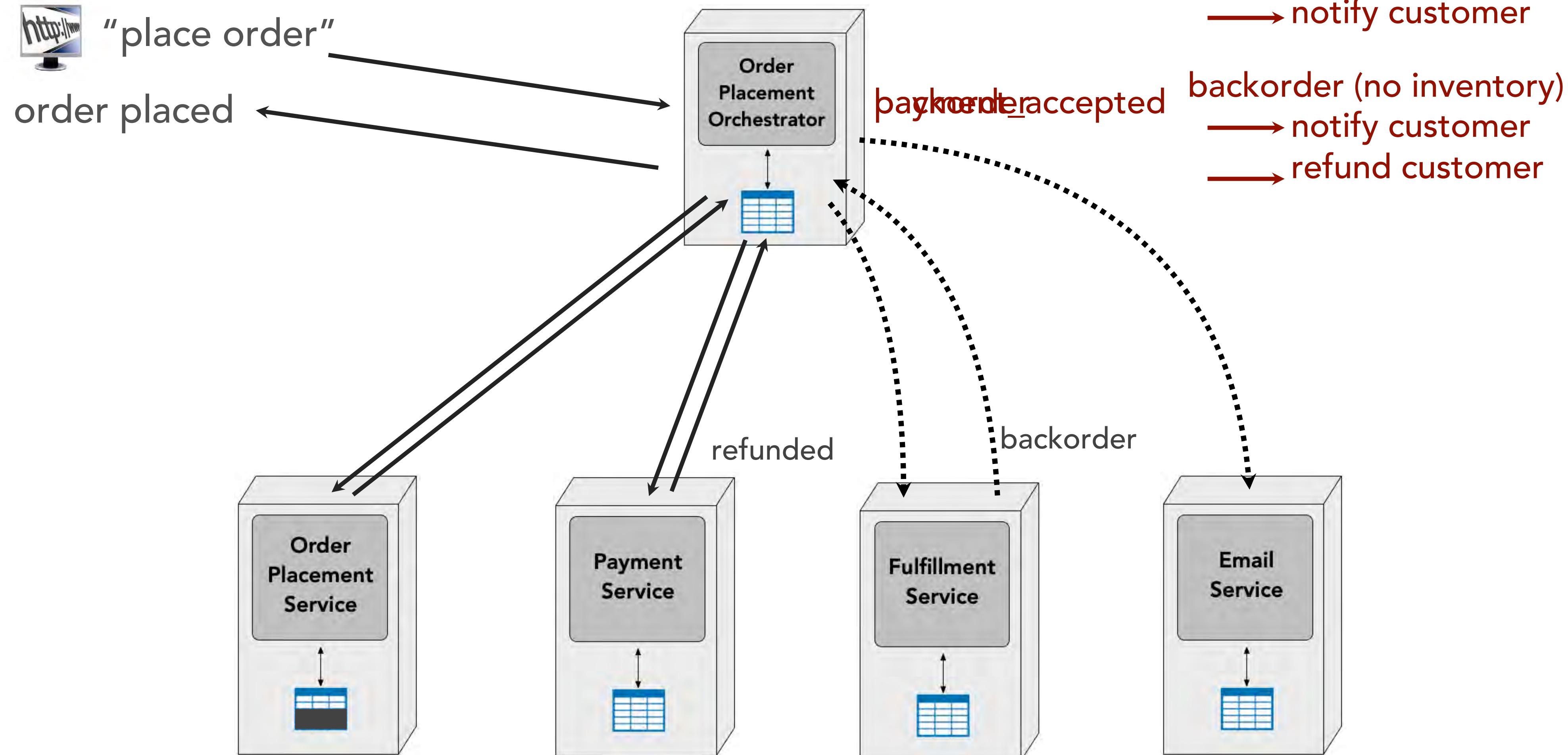
# managing workflows

## error handling



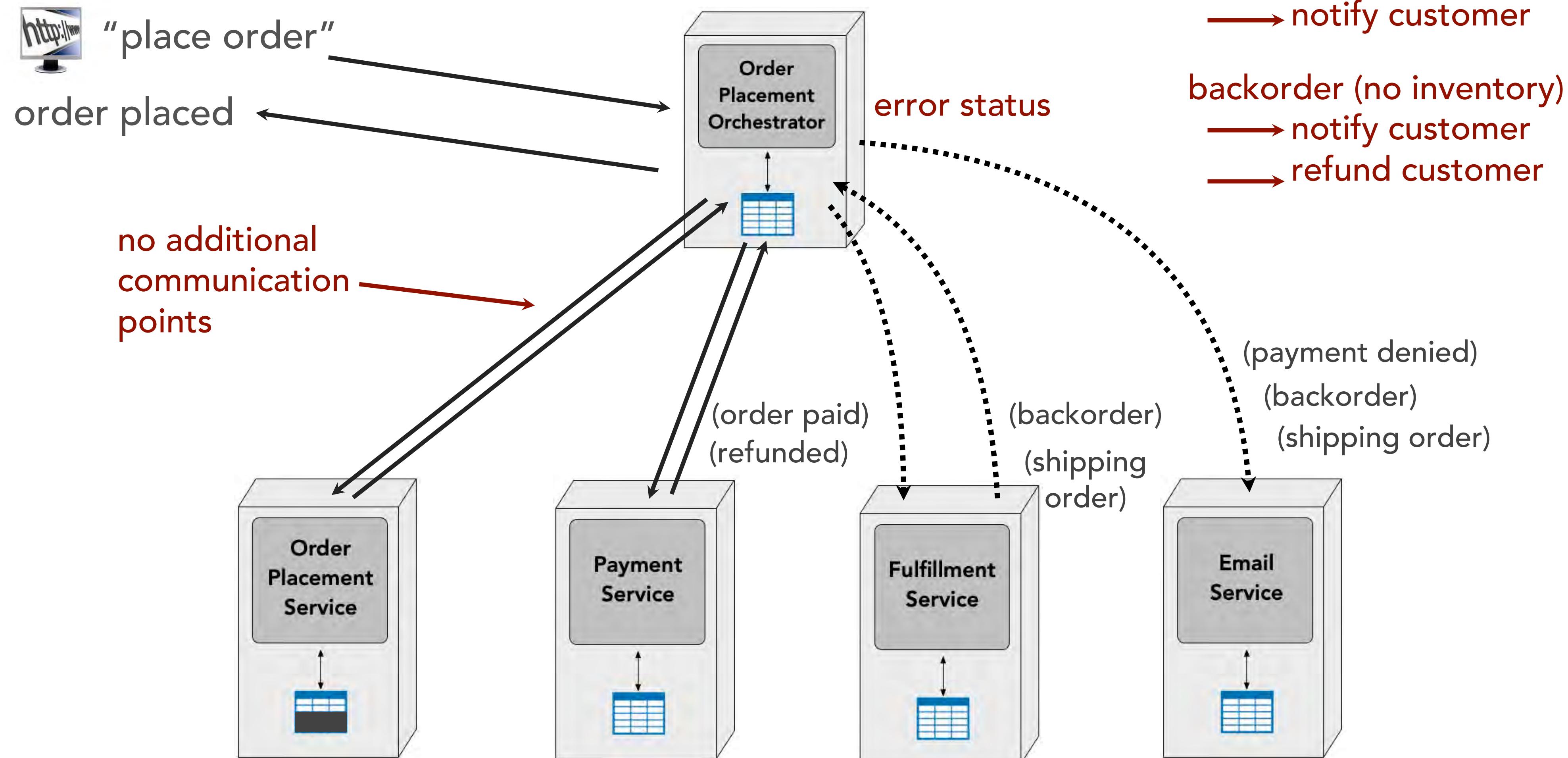
# managing workflows

## error handling



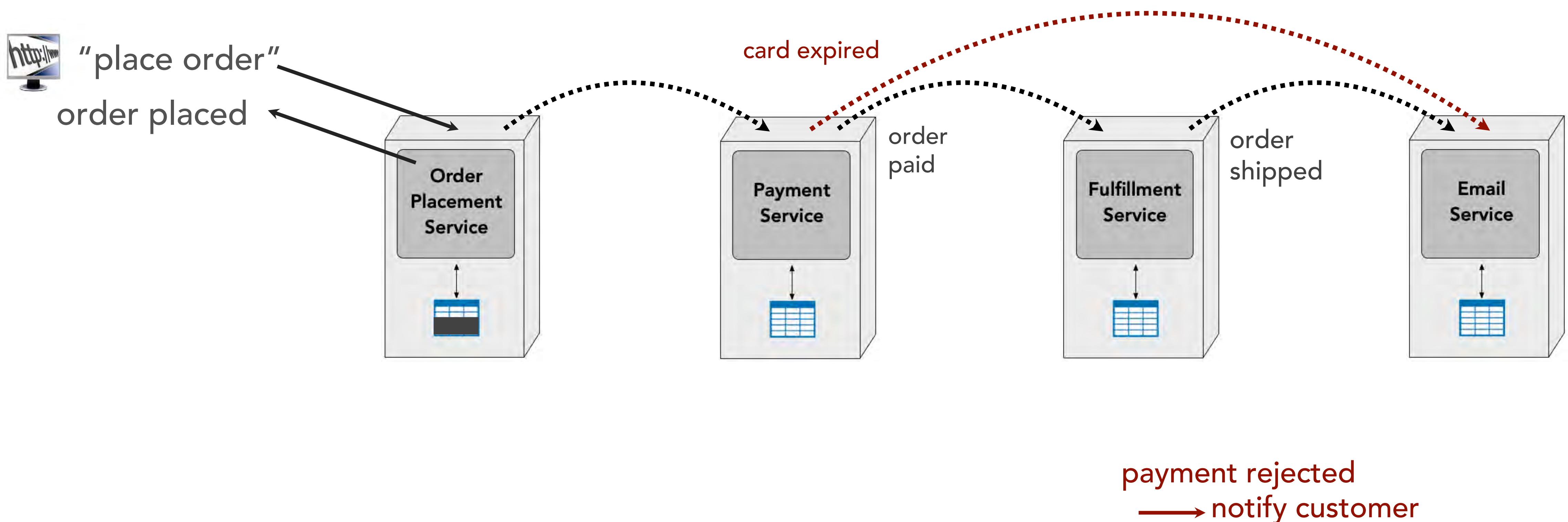
# managing workflows

## error handling



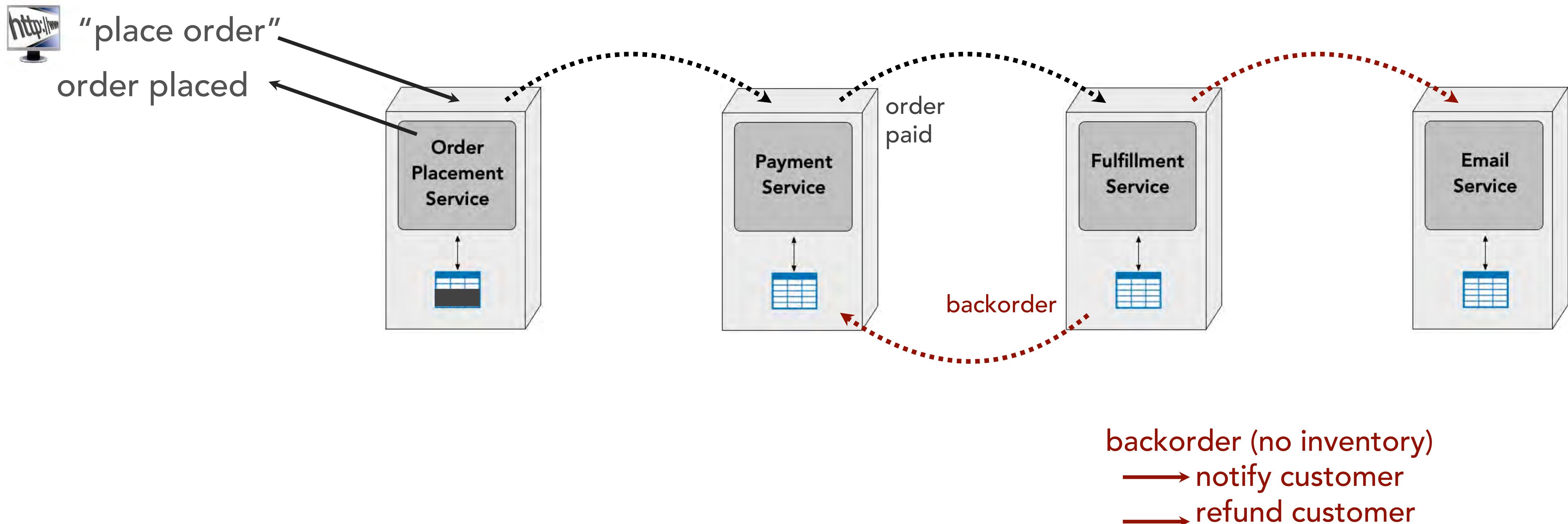
# managing workflows

## error handling



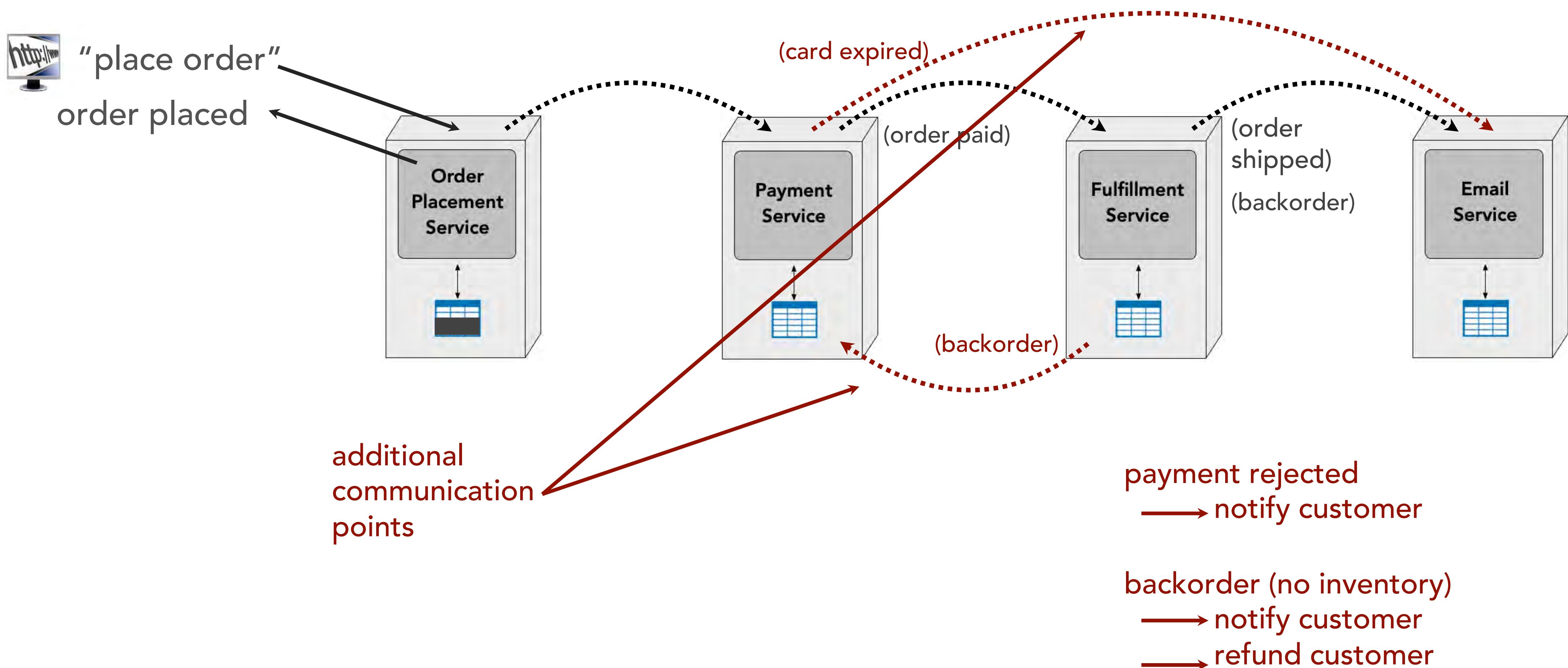
# managing workflows

## error handling

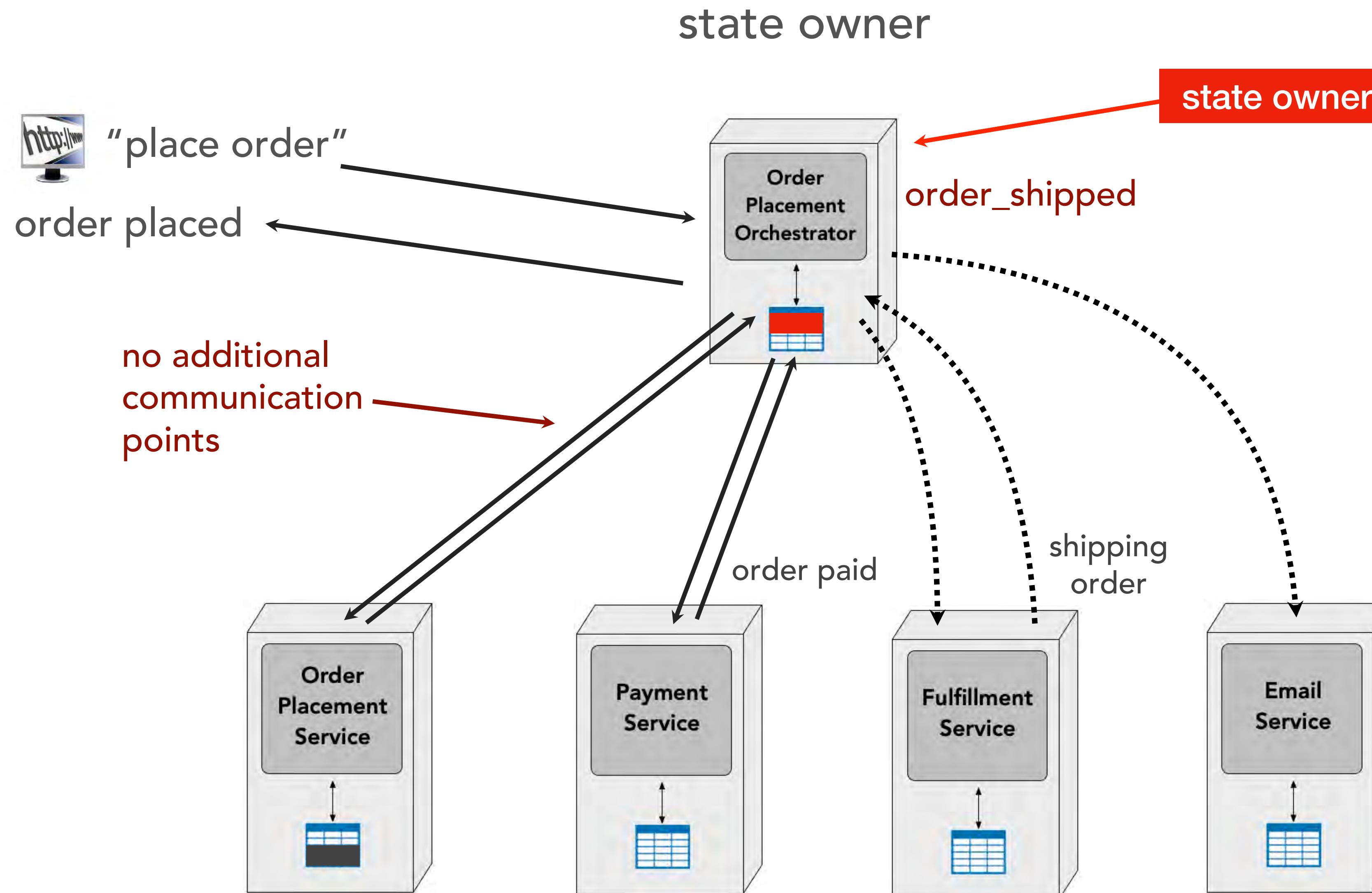


# managing workflows

## error handling

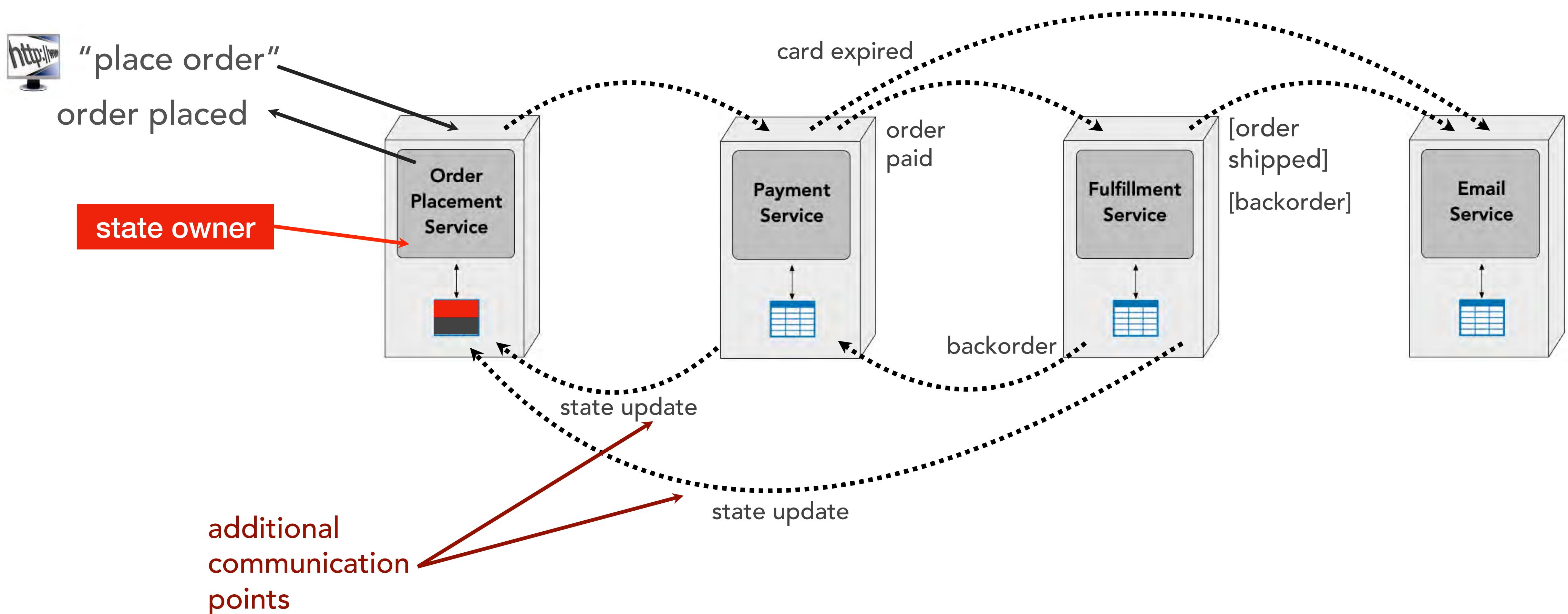


# managing workflows



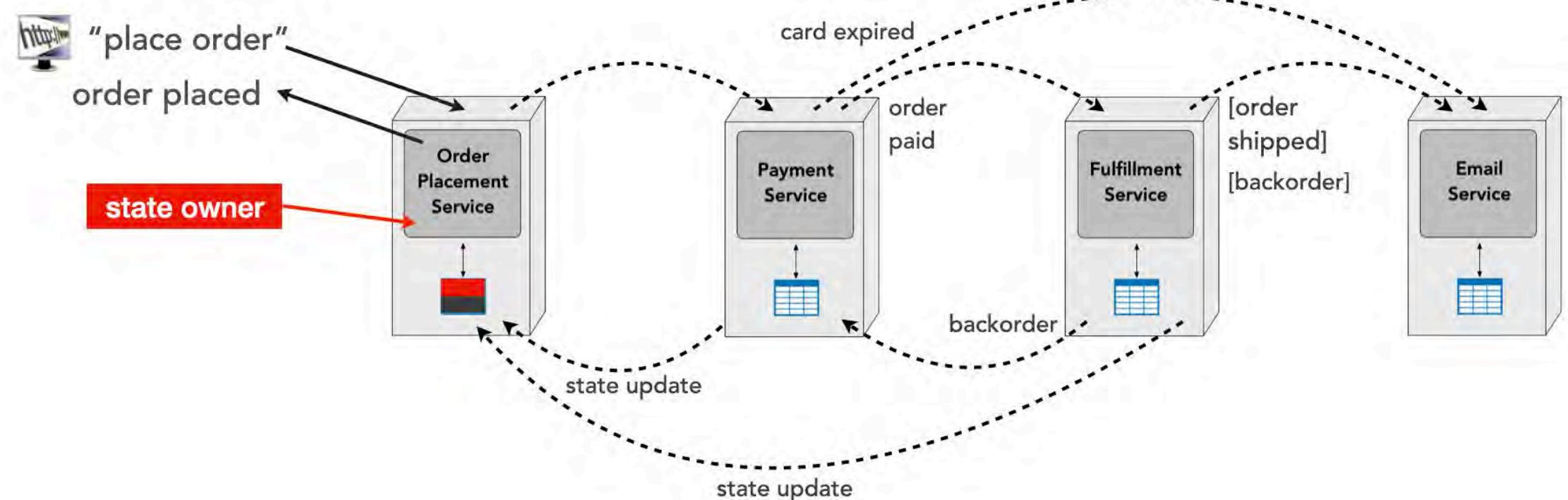
# managing workflows

state owner

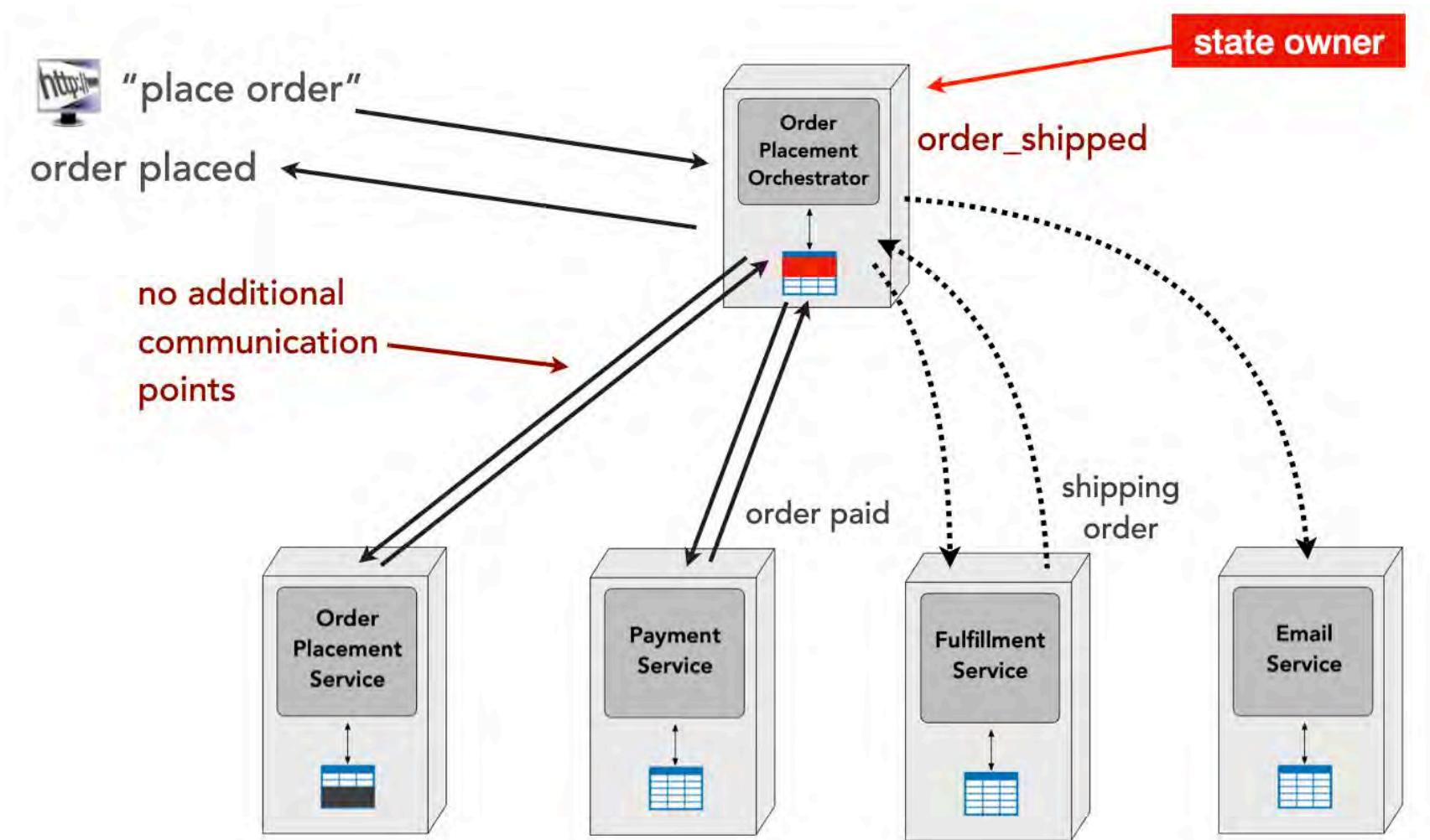


# managing workflows

## choreography

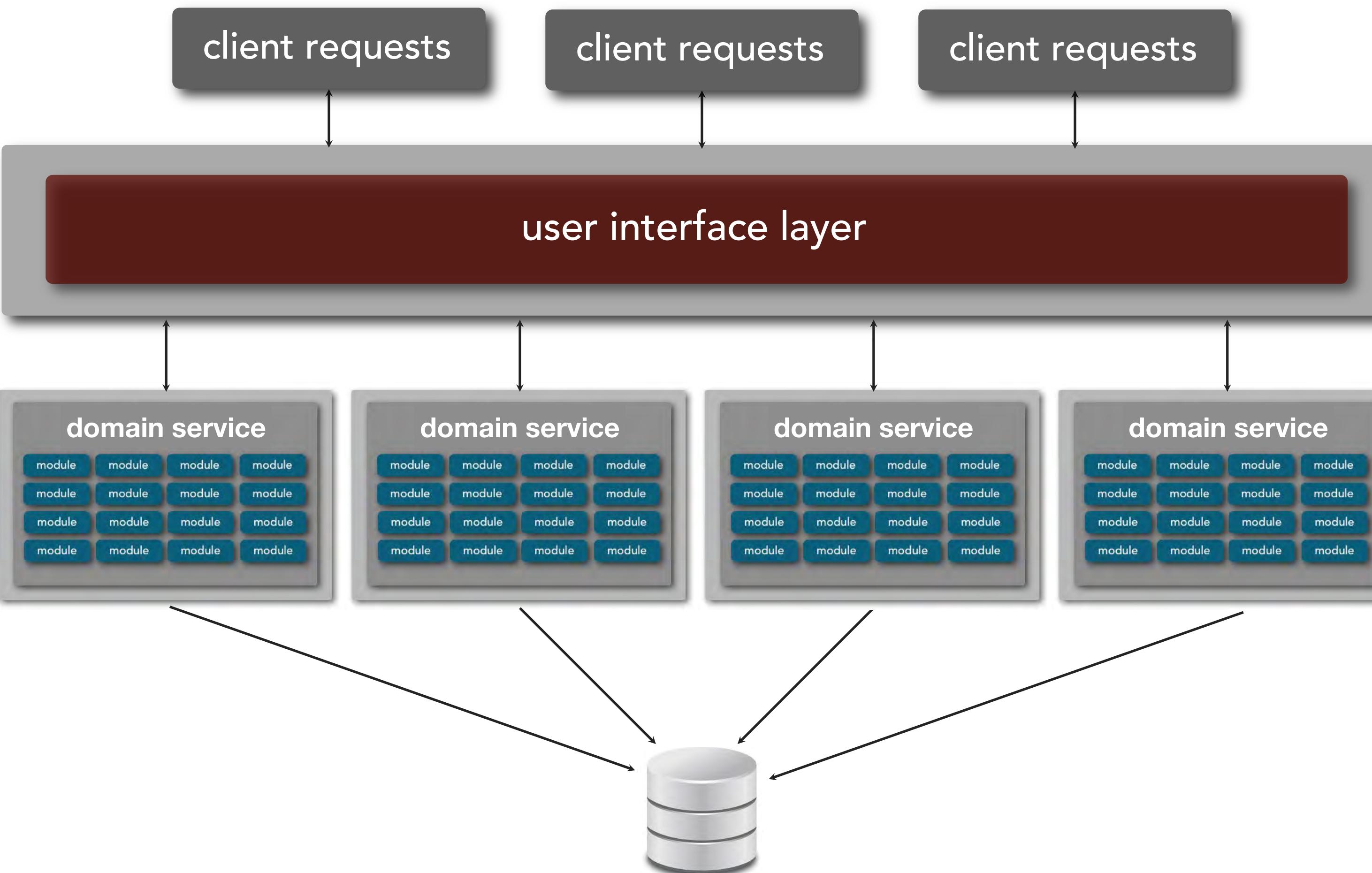


## orchestration

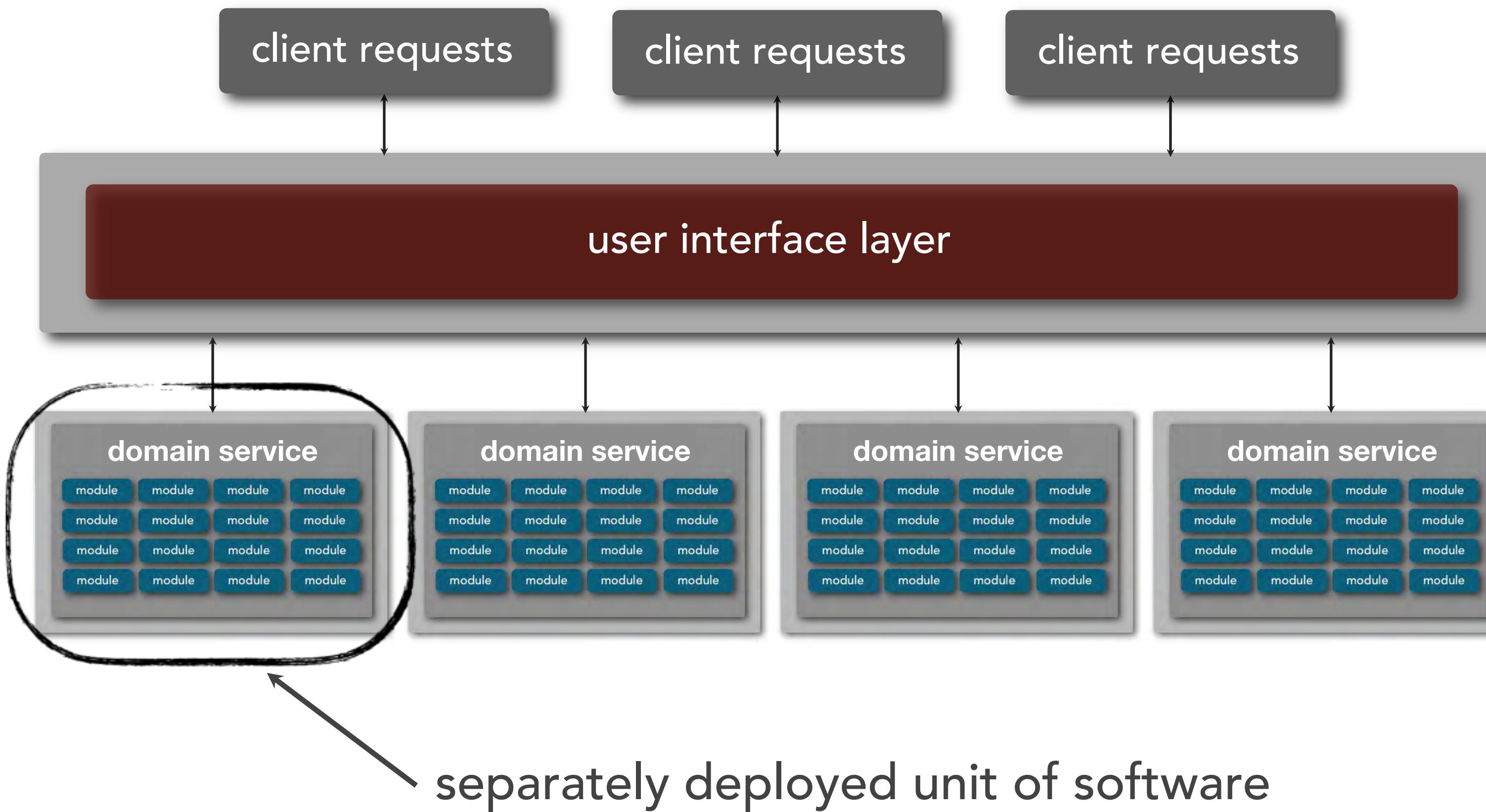


# Hybrid Architectures

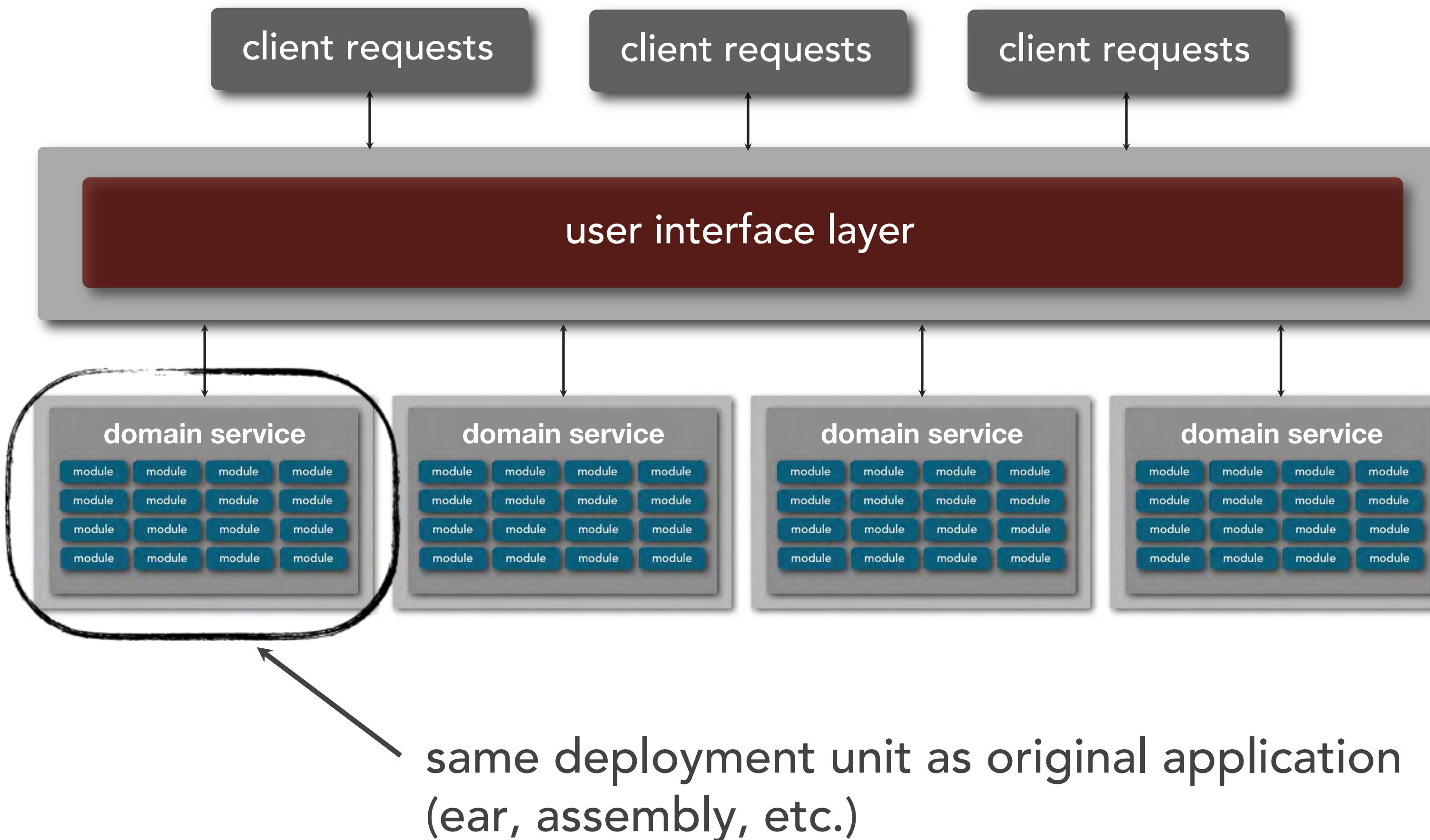
# service-based architecture



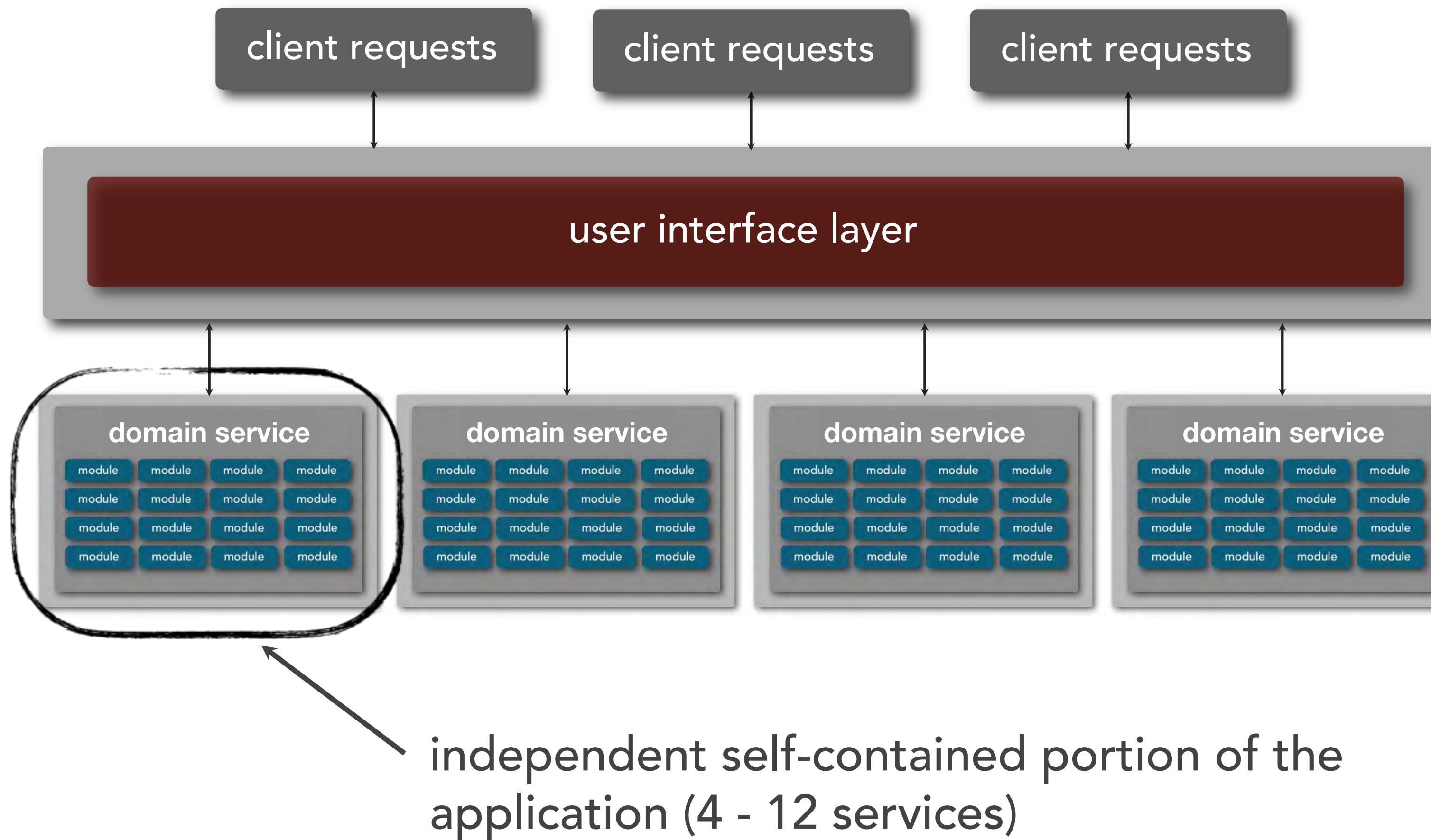
# service-based architecture



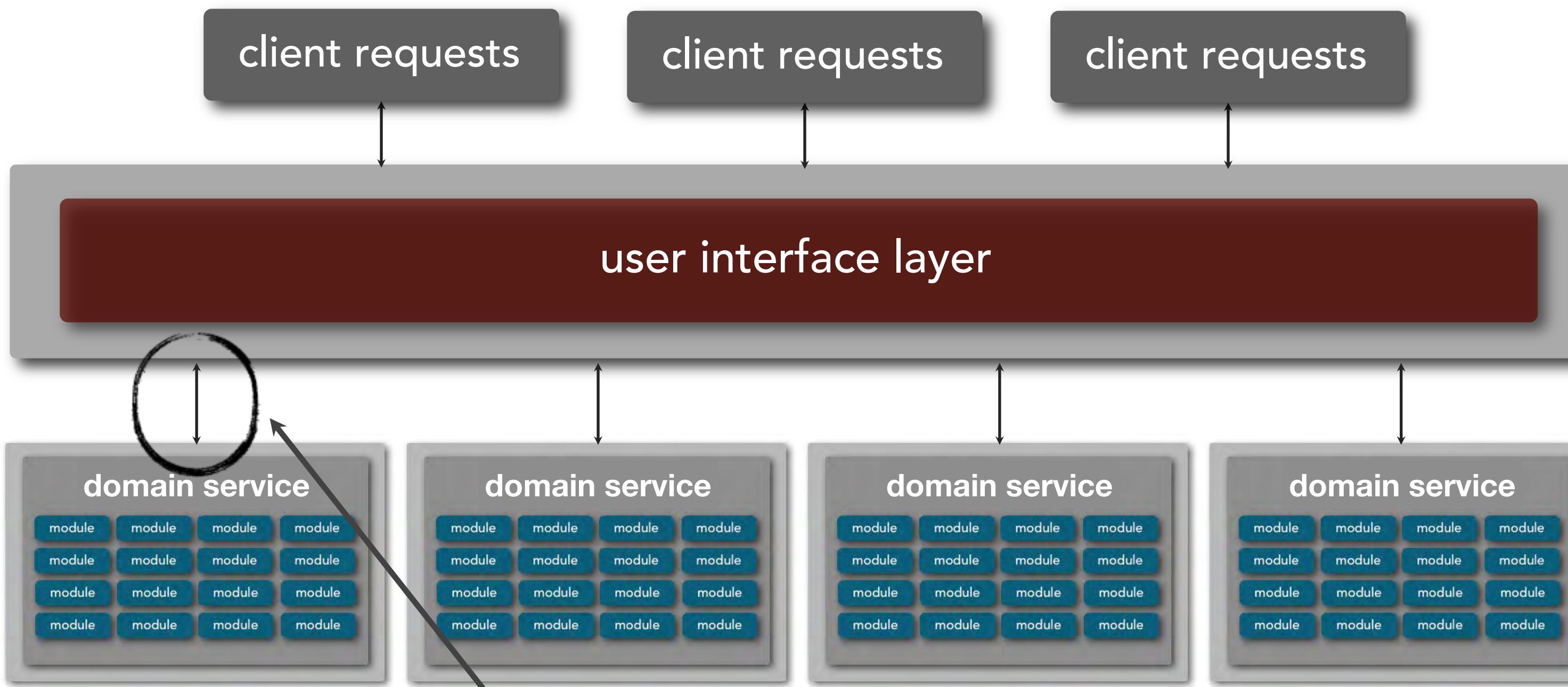
# service-based architecture



# service-based architecture

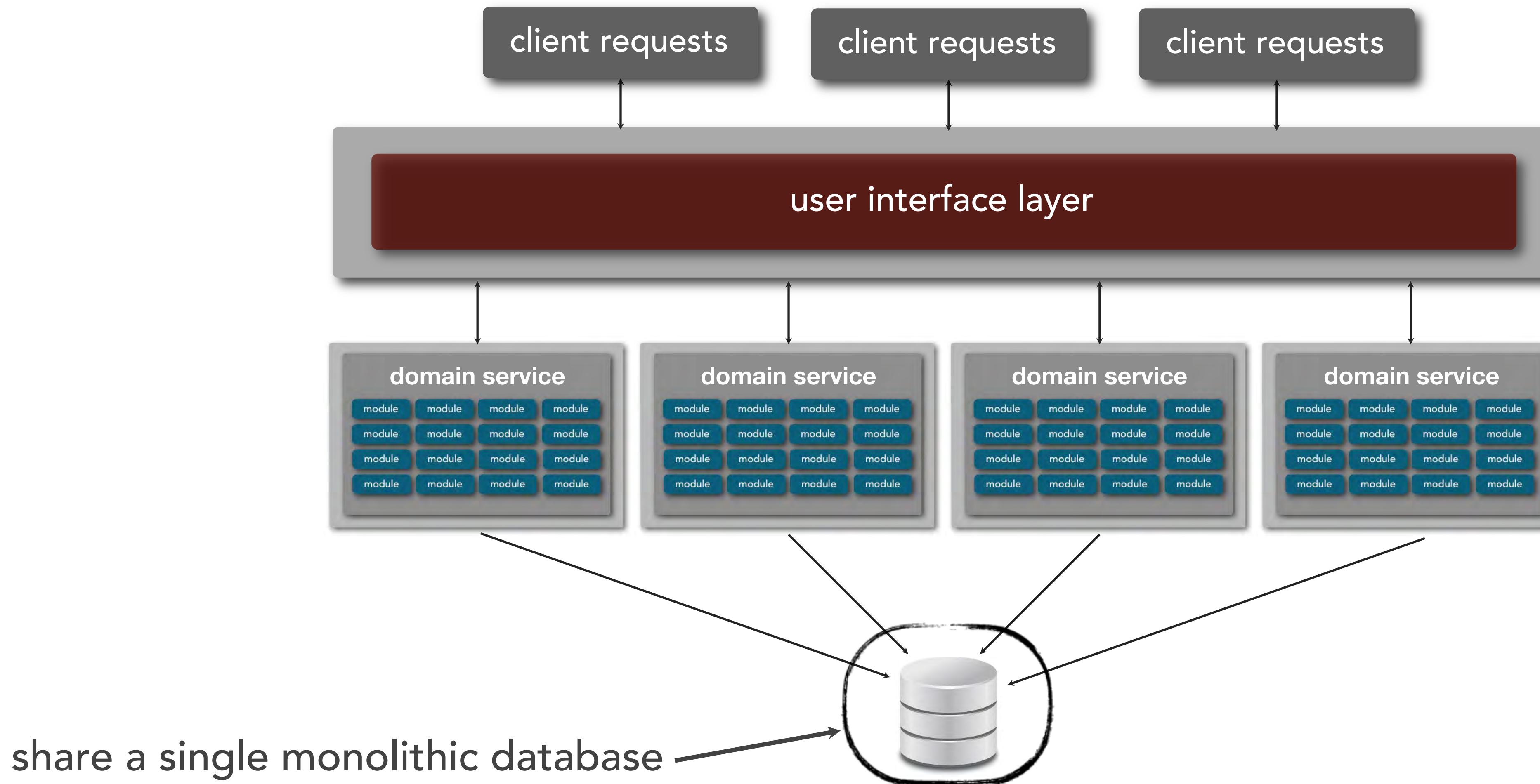


# service-based architecture

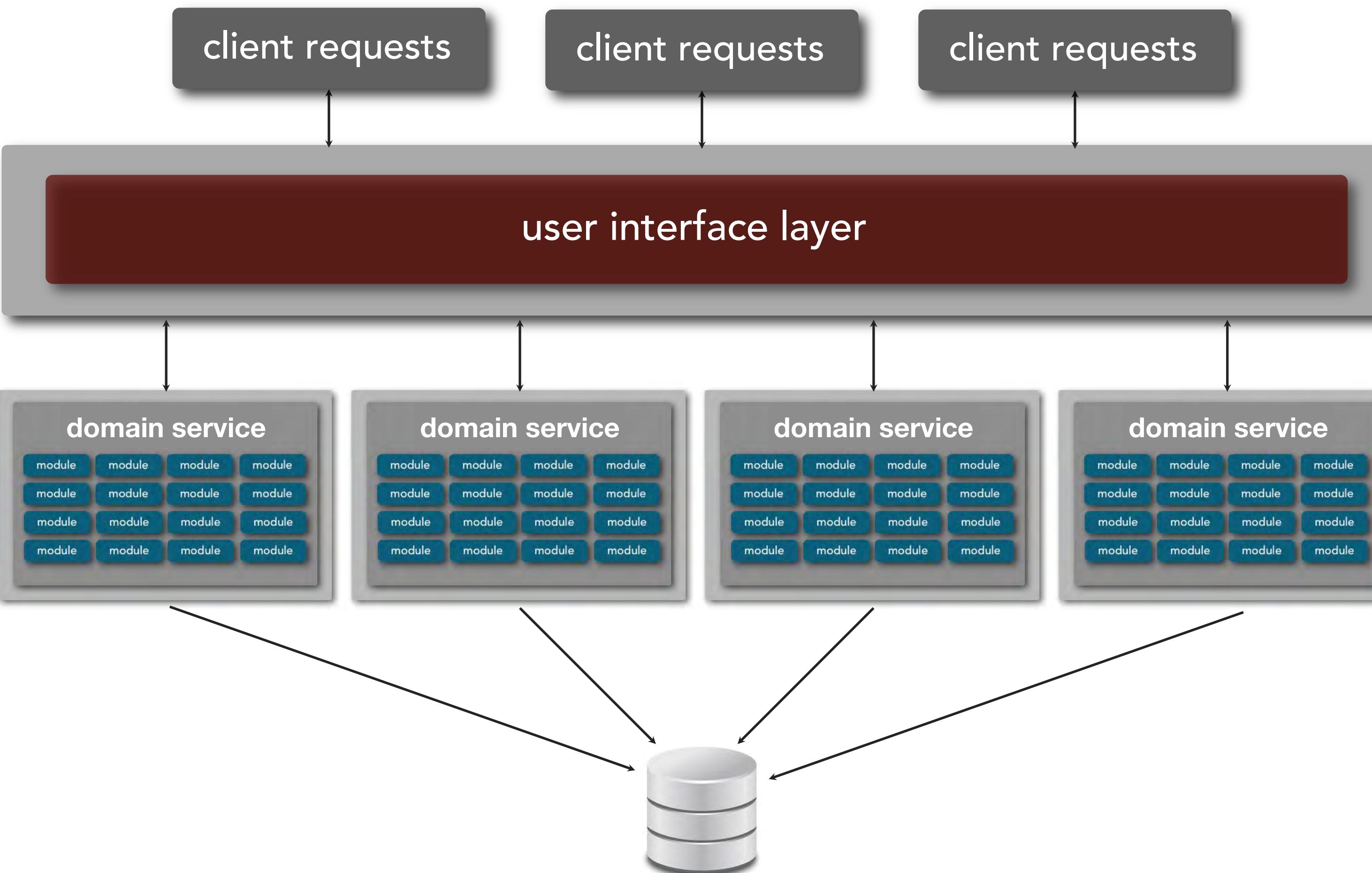


usually rest, soap, messaging, or  
remote procedure call

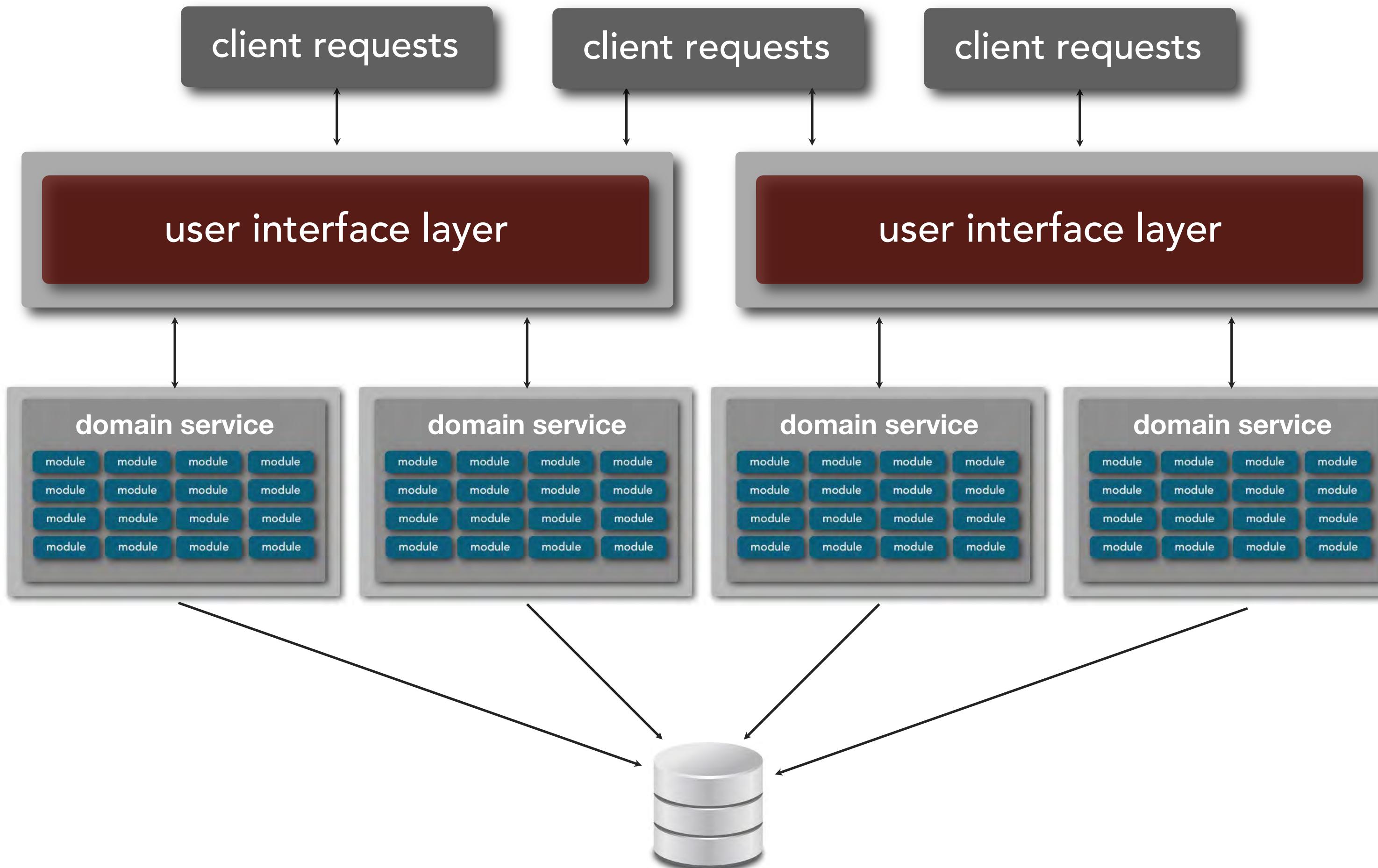
# service-based architecture



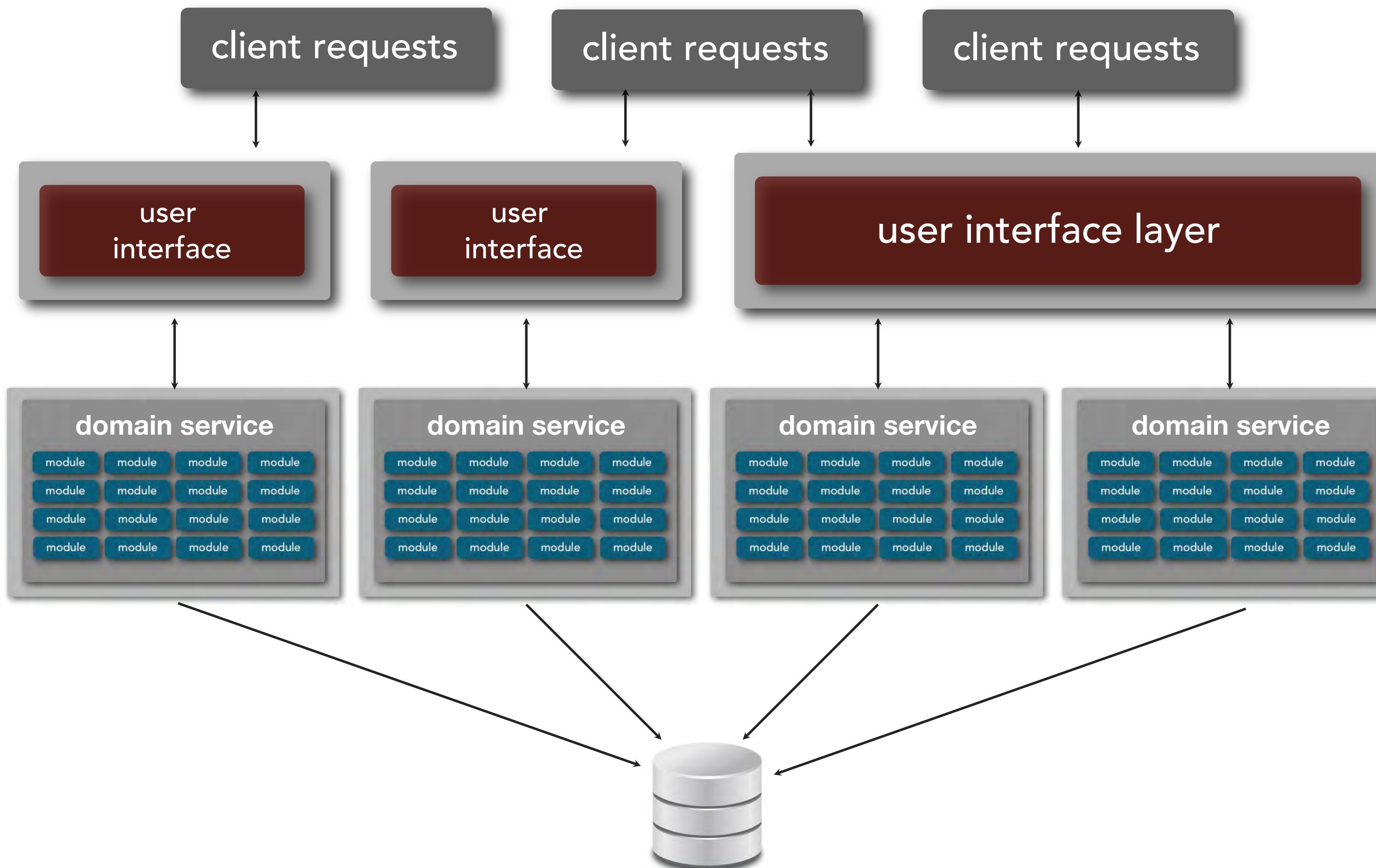
# service-based architecture



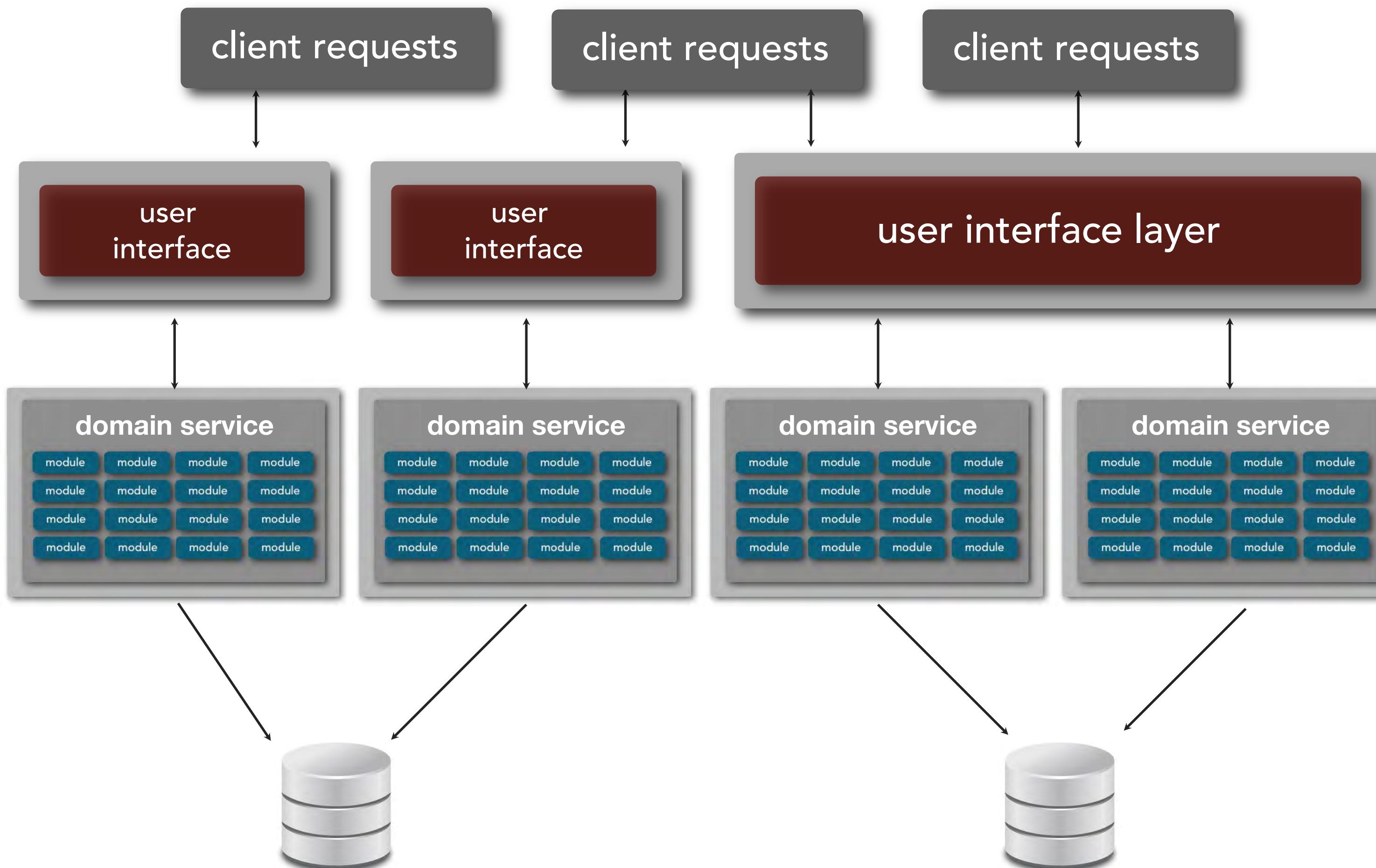
# service-based architecture



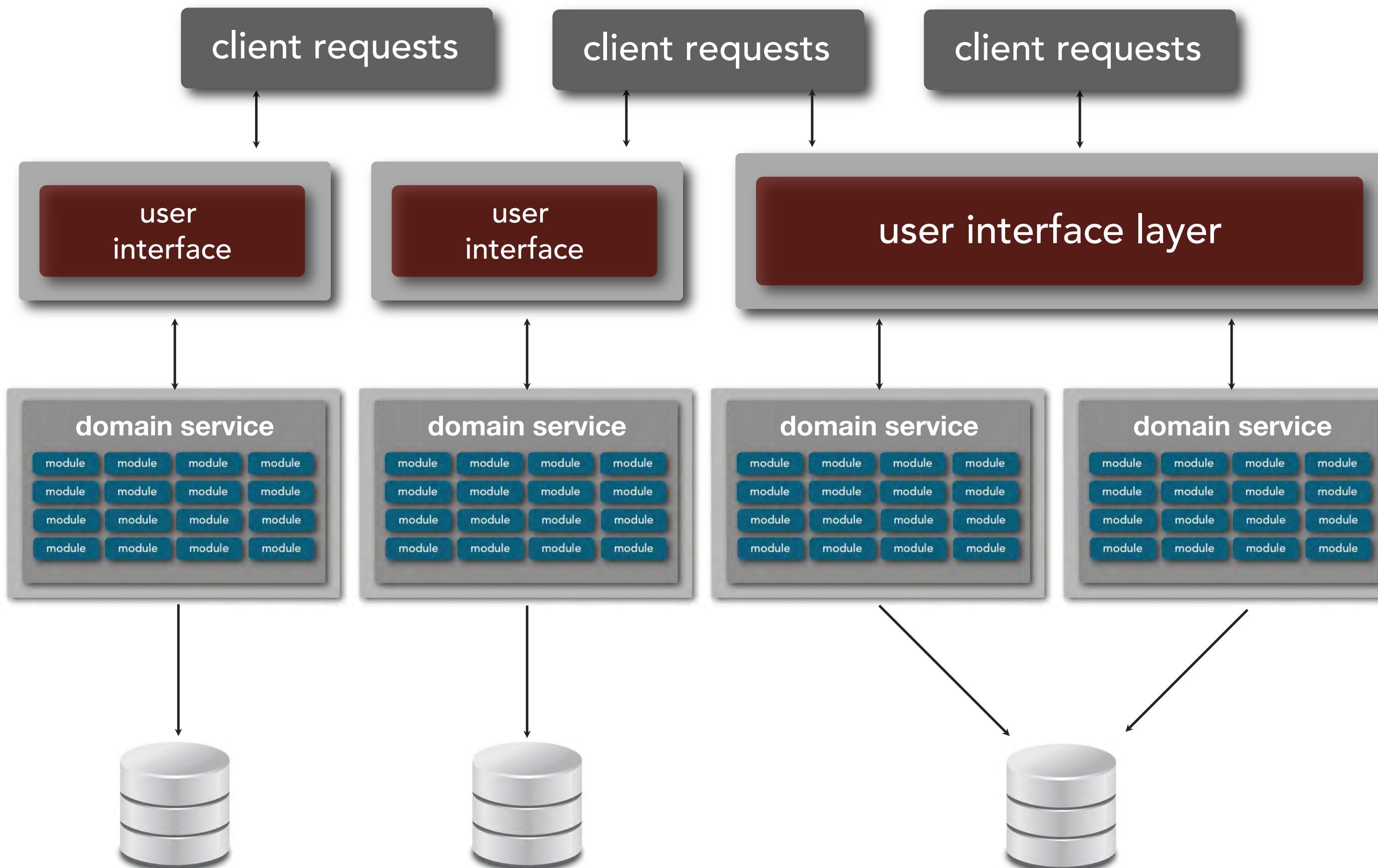
# service-based architecture



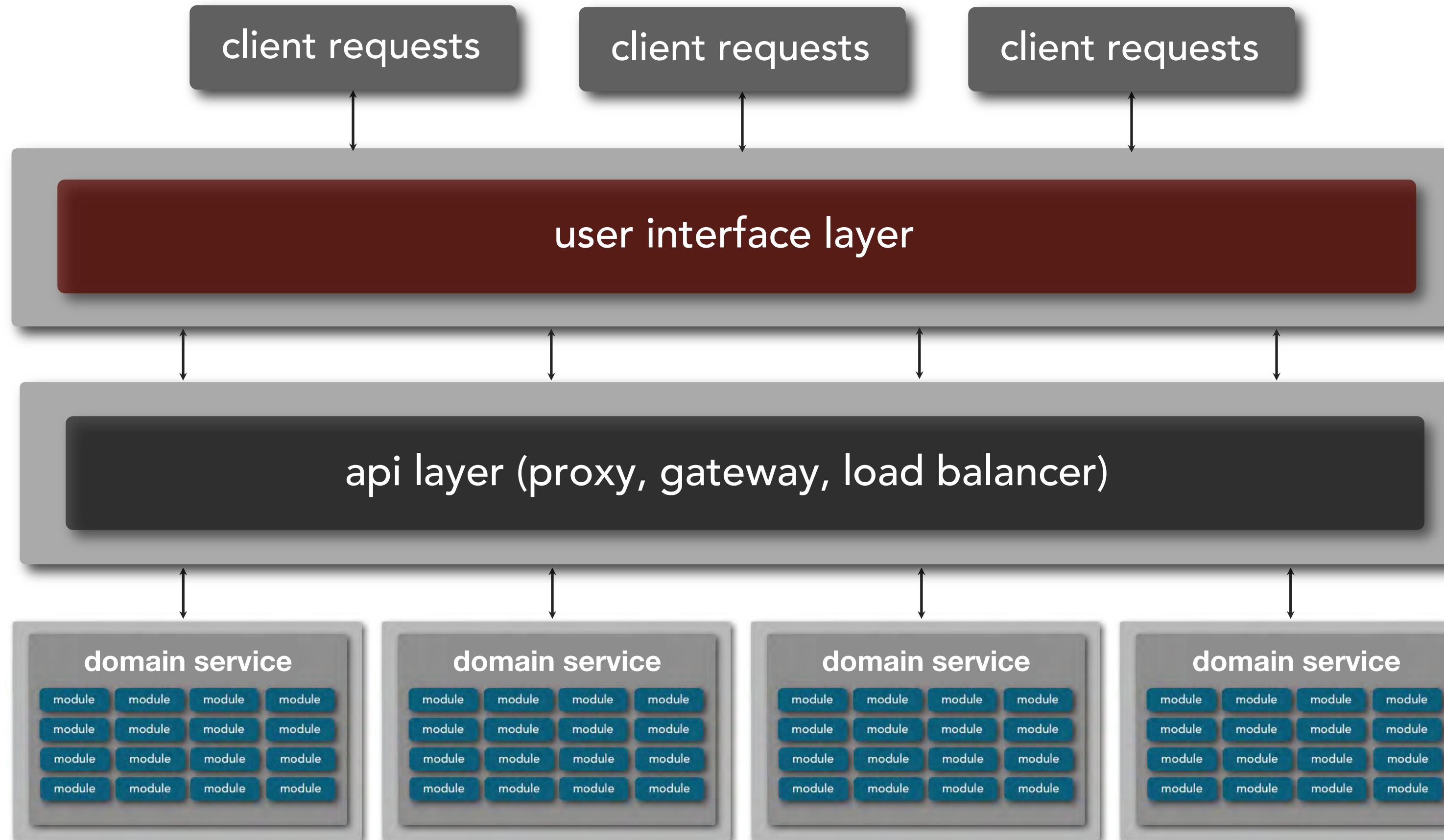
# service-based architecture



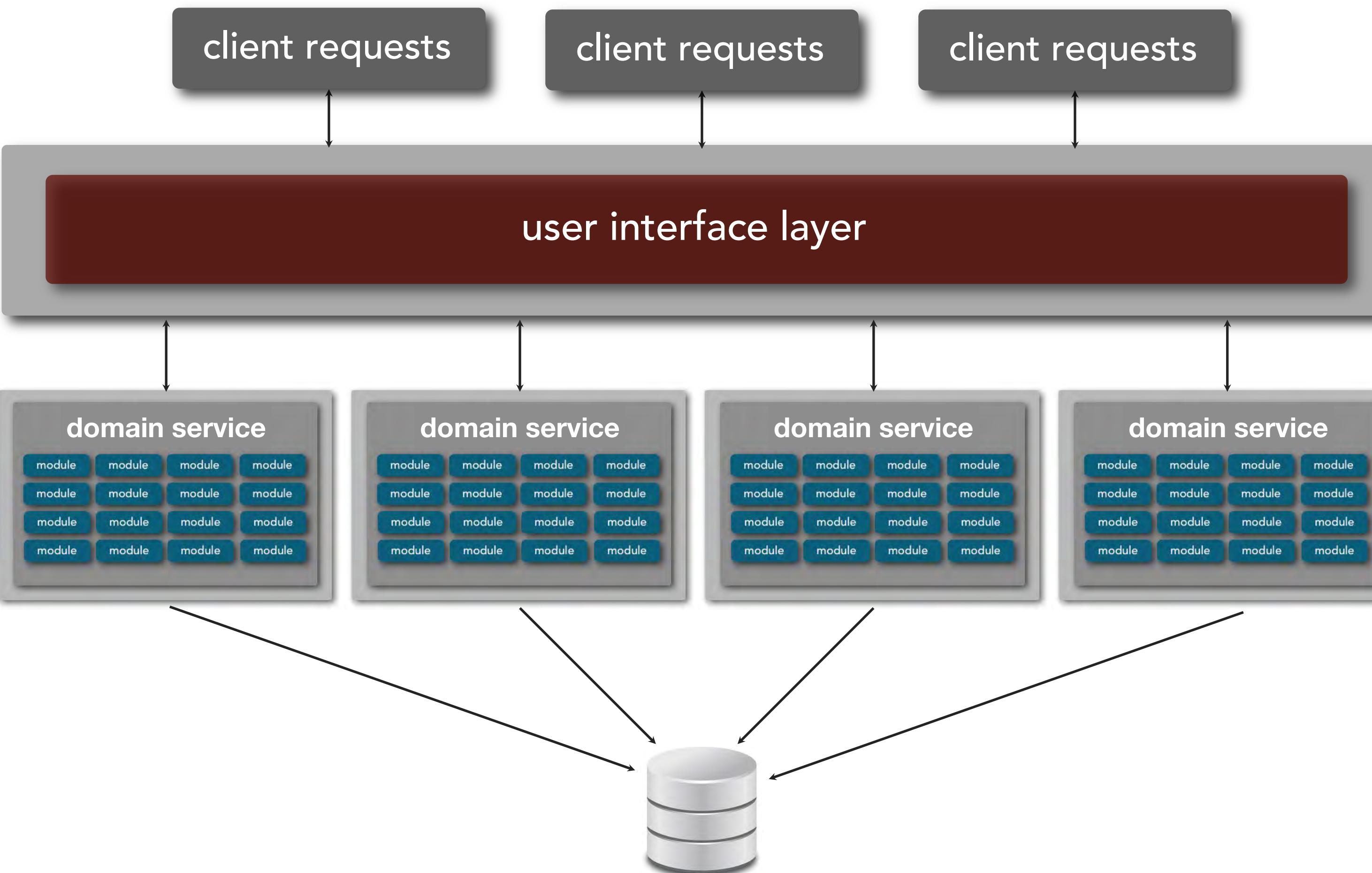
# service-based architecture



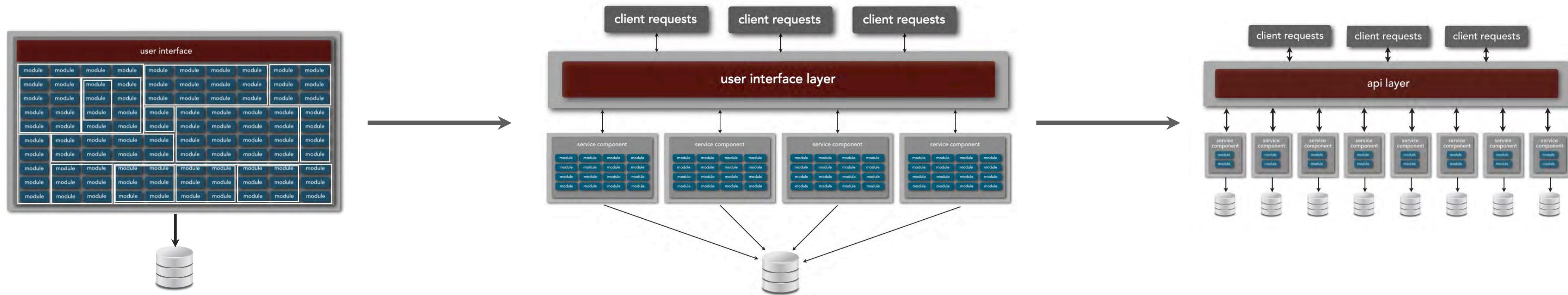
# service-based architecture



# service-based architecture



# service-based architecture

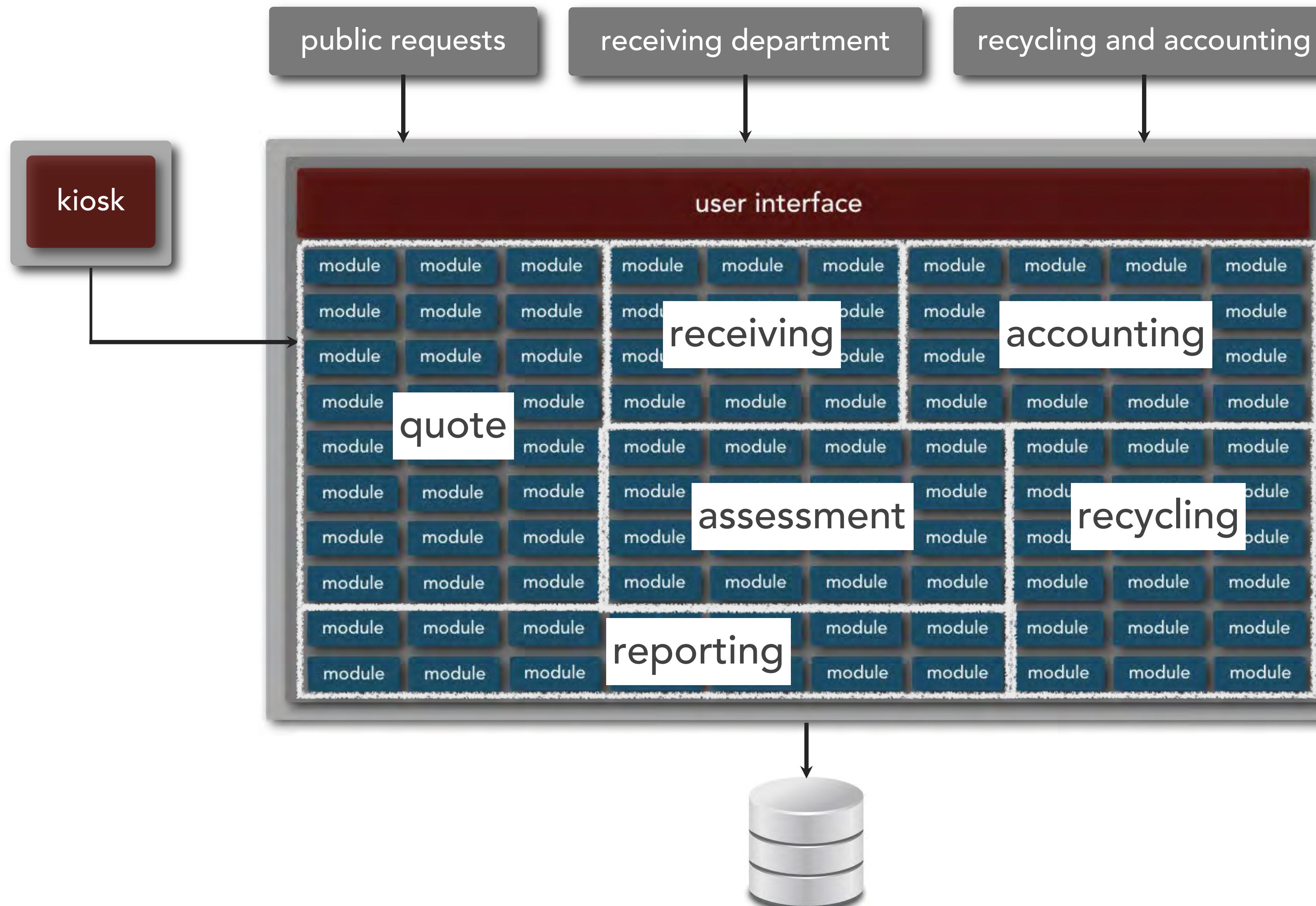


monolithic  
architecture

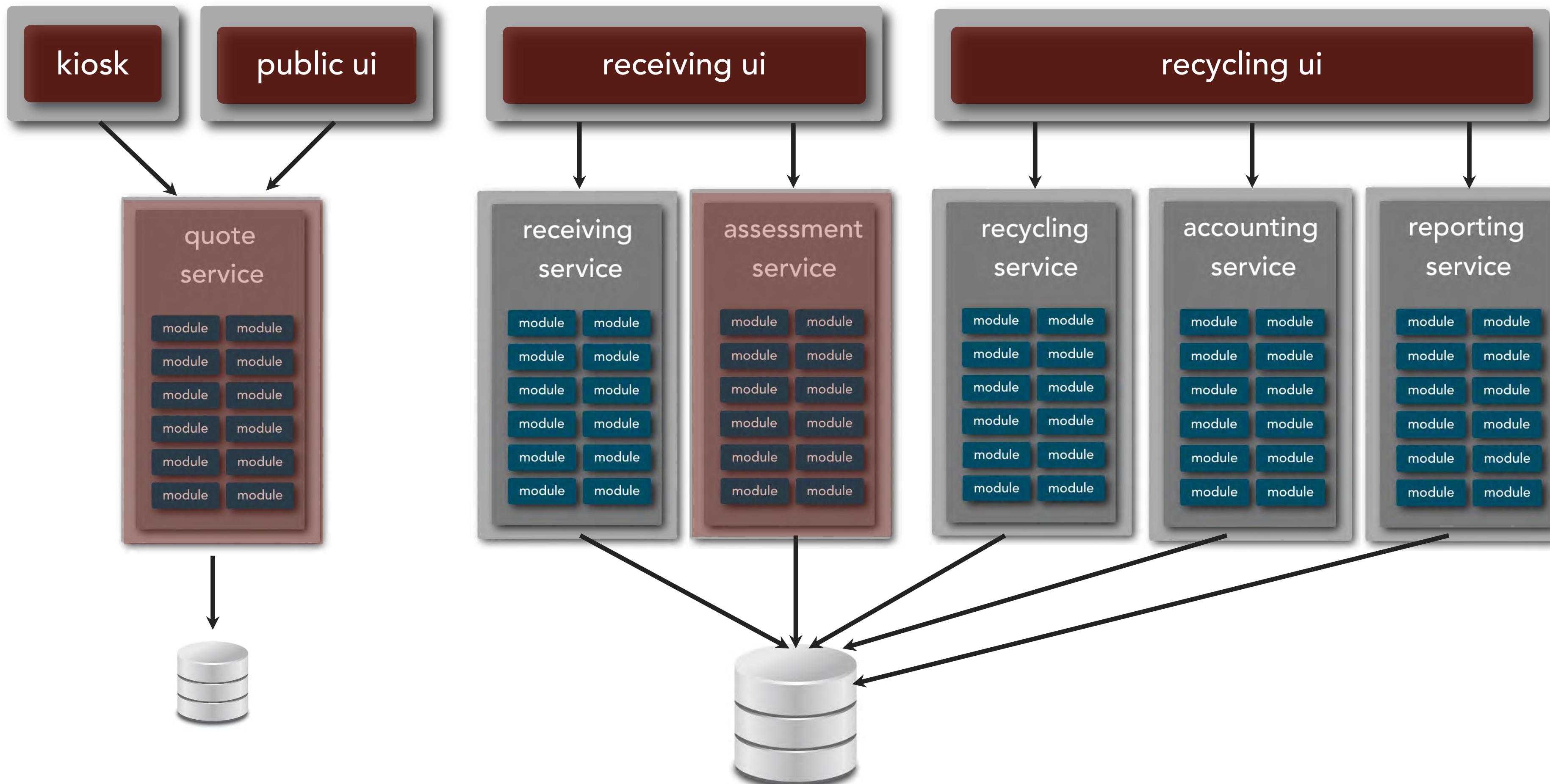
service-based  
architecture

microservices  
architecture

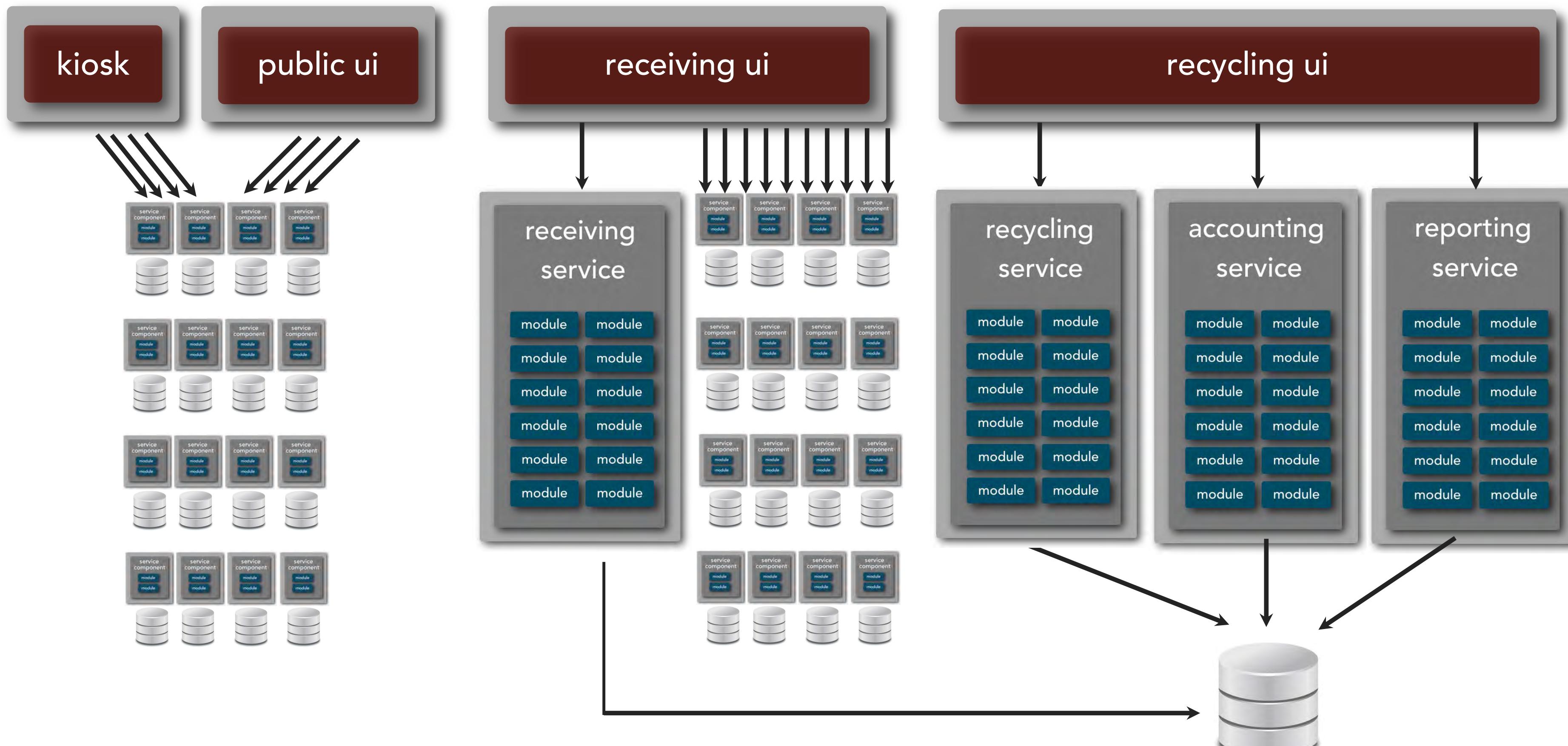
# electronics recycling application



# electronics recycling application



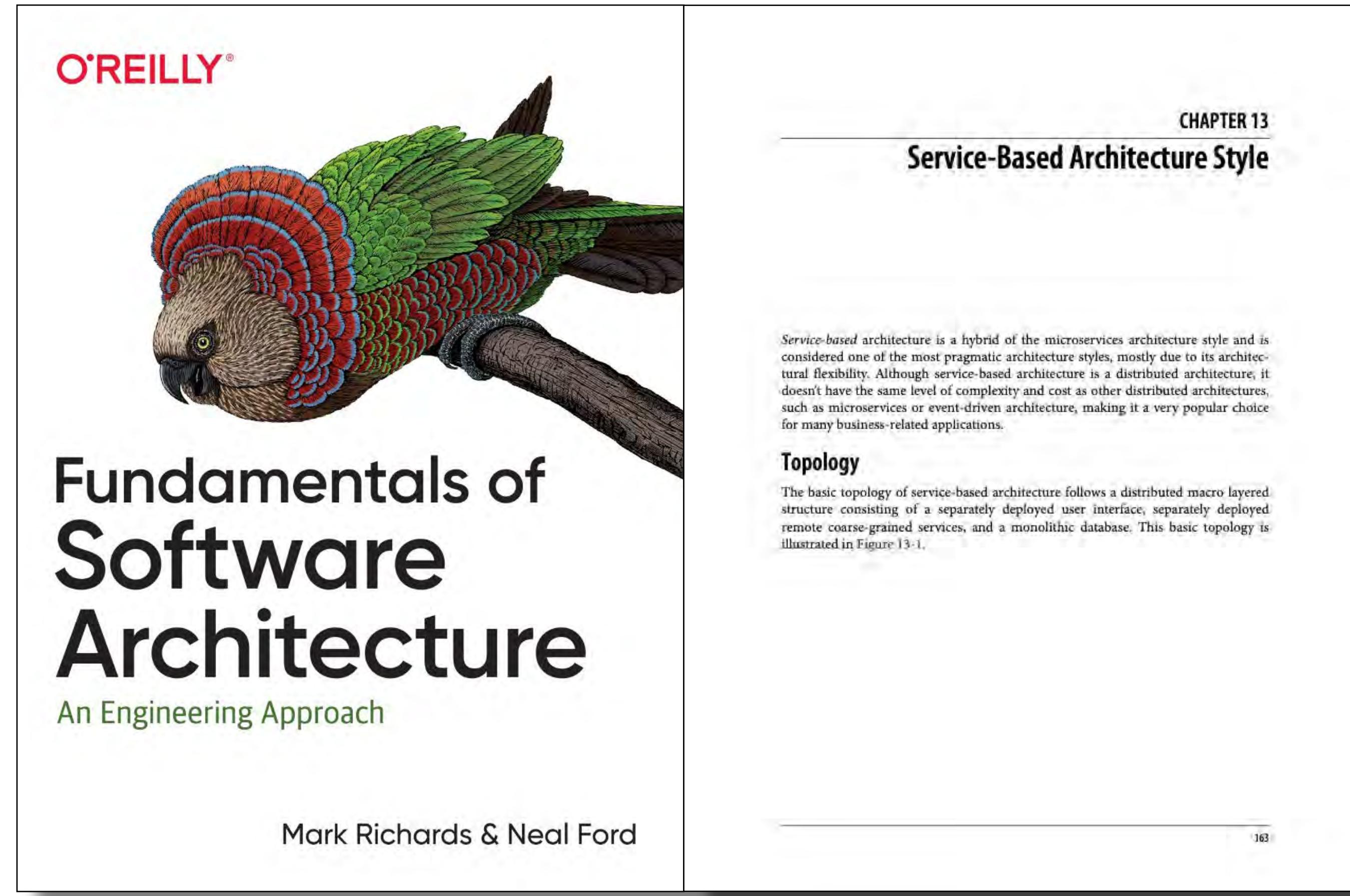
# electronics recycling application



# Fundamentals of Software Architecture

by Mark Richards and Neal Ford

<https://www.amazon.com/gp/product/1492043451>

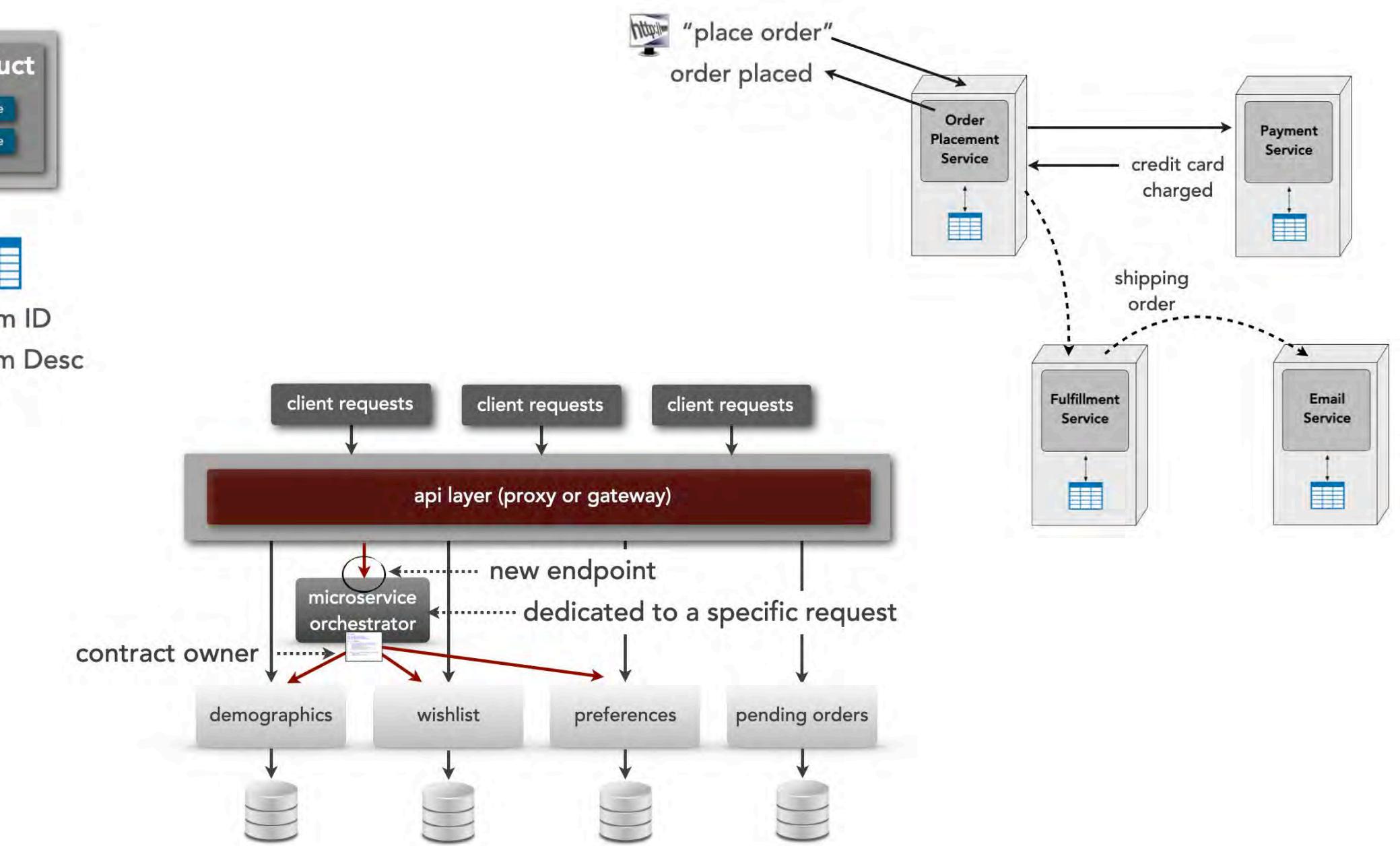
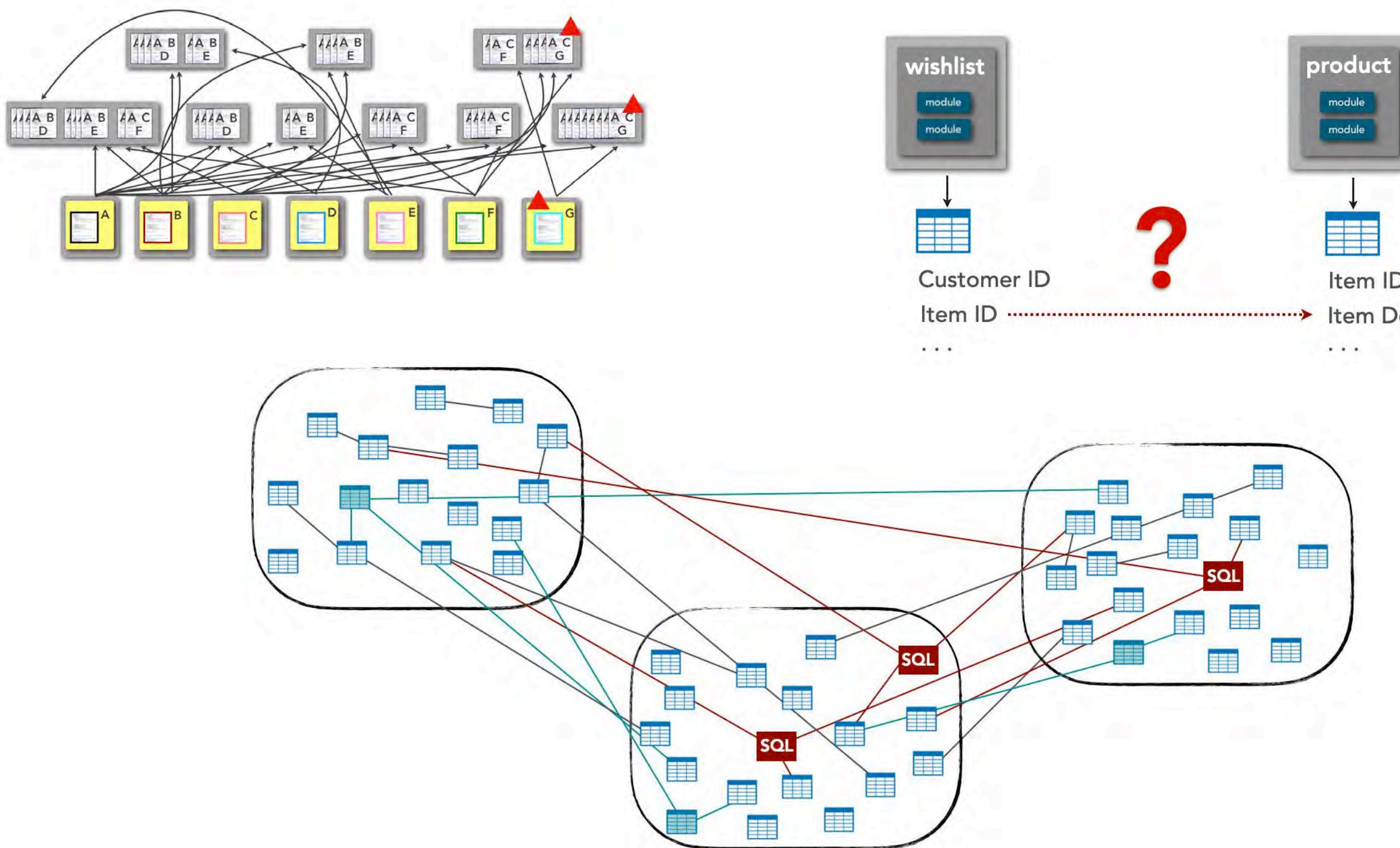


# Summary

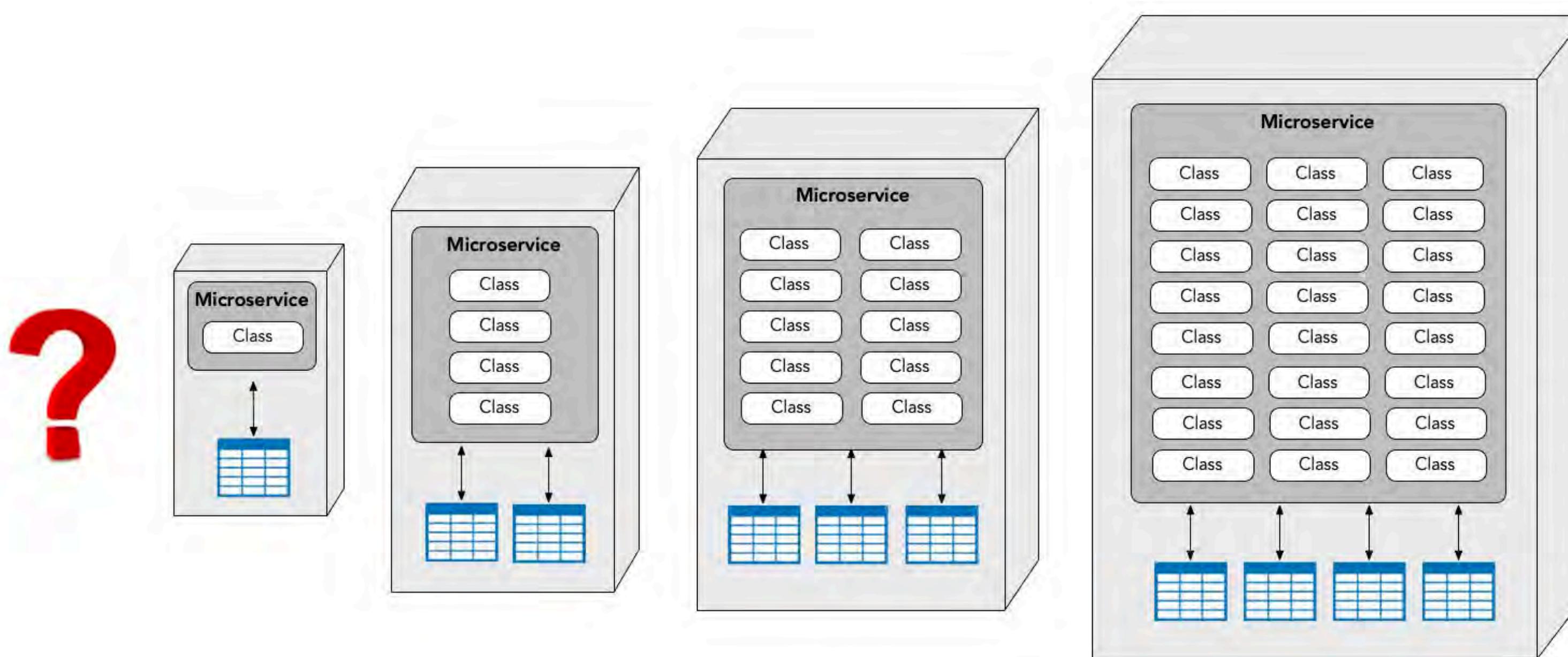
avoid jumping on the bandwagon;  
know when to use microservices  
and why



# understand the challenges of microservices and determine whether it is the right architectural fit



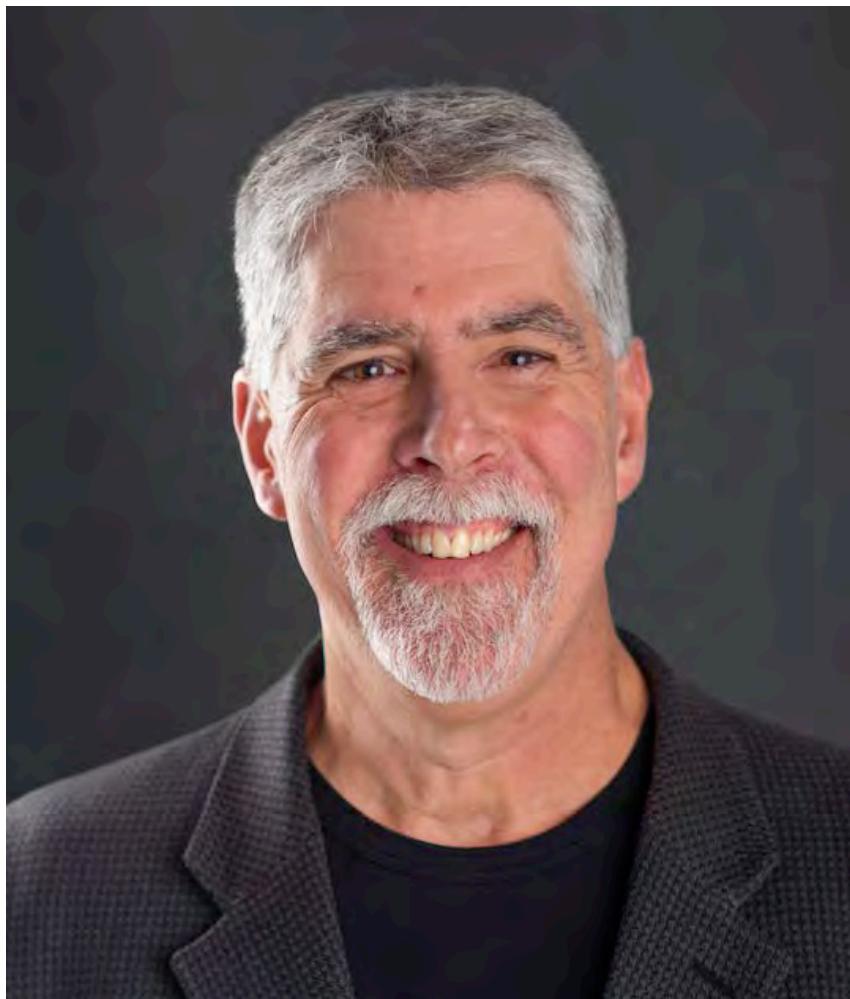
# embrace modularity but be cautious of granularity





Wi-Fi icon **LIVE ONLINE TRAINING**

# Microservices Architecture and Design



**Mark Richards**

**Independent Consultant**

**Hands-on Software Architect / Published Author**

**Founder, DeveloperToArchitect.com**

**<https://www.linkedin.com/in/markrichards3>**

**@markrichardssa**